CLAUSTHAL UNIVERSITY OF TECHNOLOGY

DOCTORAL THESIS

Web Service Based Framework for the Coupling of Simulation Models in Heterogeneous Environments

Author:

Marc WIDEMANN

First Supervisor:

Prof. Dr.-Ing. D. P. F. MÖLLER

Second Supervisor:

Prof. Dr.-Ing. Steffen STRASSBURGER

A thesis submitted in fulfillment of the requirements
for the degree of Doctor rerum naturalium (Dr. rer. nat.)
to the
Faculty of Mathematics, Computer Science and Engineering

October 2016

TU Clausthal
Clausthal University of Technology

"In the name of God, most Gracious, most Compassionate"

Acknowledgements

This doctoral thesis was written at the "Technische Informatik Systeme" workgroup of the University of Hamburg, while being employed as a research associate through the "'Airport2030"' cluster of excellence project.

I owe my lasting tanks to my advisor Professor Möller for his constant support, his professional input and the smooth supervision of my work.

I also thank Professor Strassburger for co-examining my thesis and the uncomplicated nature of our correspondence.

I would also like to thank Professor Wittmann for making my involvement in the "'Airport2030"' project possible and assisting me with his know-how.

Likewise, I want to thank Doctor Himstedt for aiding me in widening my horizon and the valuable suggestions in many discussions and conversations.

My thanks also go to Carola Tenge for always helping me with the organization of my work and all arising administrative tasks.

I naturally thank my very good friend and colleague Yousef Farschtschi, whose moral and professional support has been invaluable over the years.

My thanks go out as well to my wife, for her patience, support and help, without which I would not have been able to complete this thesis.

Furthermore, I owe gratitude to my parents, who have been supporting and encouraging me in many ways during and before my education.

Finally, I thank all my friends and colleagues from my workgroup, Doctor Beyenne, Doctor Bielecki, Doctor Drews, Doctor Güde, Doctor Hansmann, Steven Köhler, Janis Schönefeld, Daniel Sitzmann, as well as Torben Meyer from the Ilmenau University of Technology.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Equations

Abbreviations

ALSP	**A**ggregate **L**evel **S**imulation **P**rotocol
API	**A**pplication **P**rogramming **I**nterface
C&U	**C**ontainer and **U**nderground
C4ISR ITF	**C**ommand, **C**ontrol, **C**ommunications, **C**omputer **I**ntelligence **S**urveillance **R**econnaissance **I**ntegration **T**ask **F**orce
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
COTS	**C**ommercial-**O**ff-**T**he-**S**helf
CSP	**C**OTS **S**imulation **P**ackage
CSPI PDG	**CSP I**nteroperability **P**roduct **D**evelopment **G**roup
DEVS	**D**iscrete **E**vent **S**ystems **S**pecification
DIS	**D**istributed **I**nteractive **S**imulation
DLR	**G**erman **A**erospace **C**entre
DoD	**D**epartment **o**f **D**efence
FIFO	**F**irst-**I**n-**F**irst-**O**ut
FMER	**F**ederal **M**inistry of **E**ducation and **R**esearch
FOM	**F**ederation **O**bject **M**odel
GUI	**G**raphical **U**ser **I**nterface
HLA	**H**igh **L**evel **A**rchitecture
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
ID	**ID**entifier
IDE	**I**ntegrated **D**evelopment **E**nvironment
IDS	**I**teratively **D**riven **S**imulation
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IP	**I**nternet **P**rotocol

IRM	**I**nteroperability **R**eference **M**odel
IS	**I**nterface **S**pecification
ISO	**I**nternational **O**rganization for **S**tandardization
JBDS	**J**ava**B**eans **D**iscrete **S**imulation
JMI	**J**AVA-**M**ATLAB **I**nterface
LCIM	**L**evels of **C**onceptual **I**nteroperability **M**odel
LISI	**L**evels of **I**nformation **S**ystems **I**nteroperability
M&S	**M**odelling and **S**imulation
MVC	**M**odel-**V**iew-**C**ontroller
NMI	**N**C3TA reference **M**odel for **I**nteroperability
OASIS	**O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards
OIDS	**O**ptimistic **I**teratively **D**riven **S**imulation
OMT	**O**bject **M**odel **T**emplate
OSI	**O**pen **S**ystems **I**nterconnection
RTI	**R**un**T**ime **I**nfrastructure
SCF	**S**oft **C**omputing **F**ramework
SDX	**S**imulation **D**ata e**X**change
SGML	**S**tandard **G**eneralized **M**arkup **L**anguage
SIMNET	**SIM**ulator **NET**working
SISO	**S**imulation **I**nteroperability **S**tandards **O**rganization
SOA	**S**ervice-**O**riented **A**rchitecture
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
SOM	**S**imulation **O**bject **M**odel
SQL	**S**tructured **Q**uery **L**anguage
SVG	**S**calable **V**ector **G**raphics
TCP	**T**ransmission **C**ontrol **P**rotocol
TUHH	**T**echnical **U**niversity of **H**amburg-**H**arburg
UDDI	**U**niversal **D**escription **D**iscovery and **I**ntegration
UDP	**U**ser **D**atagram **P**rotocol
UHH	**U**niversity of **H**amburg

UI	**U**ser **I**nterface
UML	**U**nified **M**odeling **L**anguage
URL	**U**niform **R**esource **L**ocator
VSE	**V**isual **S**imulation **E**nvironment
W3C	**W**orld **W**ide **W**eb **C**onsortium
WS	**W**eb **S**ervice
WSDL	**W**eb **S**ervice **D**escription **L**anguage
WWW	**W**orld **W**ide **W**eb
XHTML	e**X**tensible **H**yper**T**ext **M**arkup **L**anguage
XML	e**X**tensible **M**arkup **L**anguage
XML-RPC	**XML R**emote **P**rocedure **C**all
XMPF	**X**PL **M**odel **P**ipeline **F**ramework
XMSF	e**X**tensible **M**odeling and **S**imulation **F**ramework
XPath	**X**ML **Path** language
XPL	**X**ML **P**ipeline **L**anguage
XQuery	**X**ML **Query** language
XSD	**X**ML **S**chema **D**efinition
XSD	**X**ML **S**chema **D**efinition
XSLT	e**X**tensible **S**tylesheet **L**anguage **T**ransformation

Chapter 1

Introduction

1.1 Motivation

Simulation is a capable instrument with applications in a wide number of areas. They allow to understand the conditions as they exist in a system, and improve that system by employing what-if analyses. But only a small fraction of cases where simulation might be applicable, actually make use of its advantages [BAN03]. Banks has shown that simulation is an indispensable methodology to solve problems in fields like the manufacturing industry, logistics and transportation, and service systems [BAN98]. All of the company types associated with these individual disciplines generally need to assemble a wide range of specialized know-how in order to successfully operate.

This is the where the distribution of tasks, summarized under the label of distributed collaboration, becomes an essential factor. Distributed simulation allows simulations to be designed, operated and managed by actors with geographically disparate locations and different fields of expertise, amongst other, more technical advantages [FUJ00]. Therefore, it could represent a huge asset to any collaborative effort within specific companies and even cross-company co-operations. However, the anticipated complexity and costs attributed to distributed simulation solutions and just simulation as a whole, hinders managers to realize the potential cost reductions in engineering time or department operations.

Indeed, current distributed simulation solutions are either rather inflexible because they are tailored for very specific situations, or are highly complex to allow them to be applicable in a wide array of environments. A fitting example of this problem is the HLA, being the most established state-of-the-art distributed simulation framework. Its advocates agree that it is one of the most complex standards out there, even if efforts are underway to make that complexity more manageable [STR06]. A lot of this complexity stems from the HLA's original use for military applications within the DoD, the main focus there being are as-fast-as-possible executions of simulations.

In regards to the concerns of managers of civilian enterprises described above, the emphasis of simulations in this sector should be shifted more towards simple and thus time-effective implementations above all to reduce development costs. It is therefore the author's opinion that enhancing the approachability of distributed simulation as a technology is a key factor to promote its use in a civilian setting. In an effort to work towards that goal, a distributed simulation framework has been devised that uses well-known technologies to create an alternative that favors accessibility and simplicity over performance.

1.2 Subject

The idea to develop such a distributed simulation framework sparked during work on the Airport2030 cluster of excellence project [AIR11]. Due to the expected air traffic growth in the near future, the goal of this project is to uncover optimization potential regarding the apron and terminal processes with new technologies. To this end several partners have separately implemented three models respectively simulating the transit connection to the airport, the airport terminal and airport apron.

In order to determine the global influence of all variables these models need to be coupled to create an overall simulation. This task proved to be a major challenge due to the highly heterogeneous environment. However, it quickly became apparent that the simulation would exhibit a generally unidirectional flow of information, meaning that each model would serve as data supplier for the next model, effectively creating a chain

of models [WHM09]. Therefore, a prototype was created that coupled the three models using what has been baptized as model pipelines. These manage to encapsulate the models in web services, regardless of the tools used to design them. The data, which is translated into XML, is then transferred in between the web services using pipelines.

The basic principle described here is so intuitive and relatively simple to implement, that from that prototype emerged the distributed simulation framework that is subject of this doctoral thesis. That framework has the ability to create web service shells for virtually any model, as long as it can be interfaced in some form. These web services can then effortlessly be coupled with each other to create more complex simulations. Furthermore, the framework provides different feedback mechanisms to be able to deal with simulations with a more arbitrary data propagation. Finally, it also has the ability to automatically create user interfaces for any of the model pipelines it builds.

As already mentioned, it employs established technologies like web services and XML, which have the advantage of being supported in a lot of systems and thus are applicable in a wide array of different environments, obviously including the WWW. Indeed, the framework has the benefit of creating distributed simulations that are natively executable and manageable from the Internet through their user interfaces using any standard browser software, which is arguably a very valuable trait in any distributed collaboration environment.

1.3 Approach

To adequately describe and analyze the model pipeline framework developed in the context of this thesis, it is important to begin with discussing the basics and current state of the distributed simulation research. The second chapter will begin by briefly outlining the history of the subject, and the parallels to distributed collaboration. This will then serve to identify and study the pivotal qualities when working in distributed environments. Afterwards, this thesis will present an overview of different architecture

principles to help gain insight into the inner workings of the HLA, which is the currently dominating state-of-the-art distributed simulation architecture, and of course of the model pipeline framework.

The main objective of the third chapter is to study the inner workings of a model pipeline, using the example of the prototype designed for the aforementioned Airport2030 project, which is successfully used to couple three disparate models into a composed, more complex simulation. At first, the project specifications are explained. Then the processes of interest for the individual models are identified and described. This leads to a discussion on how to couple these models, and why the pipelining schema was a fitting approach. Afterwards follows an in-depth description of the employed models, how the model pipeline prototype handles the coupling on a technical level, and how to execute the overall simulation created by it. The chapter is then concluded with a passage demonstrating the capabilities of the web-based interaction functionality of the prototype.

Chapter 4 deals with the feedback mechanisms of model pipelines and simulations in general. After a few basic concepts and their related notions will be laid out, the chapter will describe a few commonly used synchronization techniques as a reference for the follow-up, which consists of a comparison between classical and pipeline coupling mechanisms. Afterwards, the two feedback algorithms that can be used in model pipelines will be examined and subsequently compared by using the already acquainted prototype as testbed. This will help determine the application scenarios which are best suited for either of them.

Chapter 5 will then dive into the description of the actual model pipeline framework. After the characteristics of the runtime environment have been explained, an in-depth analysis of the web service and web interface structure will follow. The uncomplicated nature of the operation of the framework stems from the ability to direct the automated process of building pipelines simply by defining the different data types used in a simulation and its involved models. The chapter is therefore concluded with the specification of the various data types available in the framework, an explanation on why these are

sufficient to build model pipelines, and instructions on how to operate the framework with them.

Chapter 6 is used to demonstrate the validity and applicability of model pipelines. It achieves it by implementing an exemplary transfer of an HLA simulation to a model pipeline one. The chapter starts with an in-depth description of the processes involved in the execution of an HLA federation. Thereafter follows the specification of the particular HLA federation the author has chosen as testbed. It simulates in a basic fashion the processes involved in the transport of goods in a dry port scenario. The example is rather simple but manages to cover a lot of the HLA functionality. The process of realizing this simulation as a model pipeline is then documented. Finally, simulation runs of both implementations and their evaluation will aid assessing the two approaches and determining possible discrepancies between them.

Finally, in the last chapter, this thesis will end with a summary of the previously discussed subjects, an analysis of the obtained results, and a conclusion if the goal to create a highly usable and flexible distributed simulation framework was brought to fruition. Furthermore, an outlook will be presented recommending the direction of further research concerning the model pipeline framework.

Chapter 2

State-of-the-Art of Distributed Simulation

2.1 Fundamentals

Simulation and its distributed execution has been the subject of research for nearly five decades. The following section will recapitulate the major achievements and historic milestones of that time frame, as well as the advancements that the research has brought. Furthermore, the relevance of distributed collaboration in the appropriate context will be explained.

2.1.1 Distributed Simulation

Discrete event simulation was successfully used in the development of small applications since the early 1960's [TOC63]. Since then, the modeled systems did not stop growing in scale and complexity. Enabled by theoretical research and advances in network computing, distributed simulation emerged in the late 70's, Thorpe generally being credited with the original concept [COS95]. Distributed simulation handles the execution of simulations on loosely coupled systems, including geographically distributed computers interconnected by network technology. The benefits of this technology are

substantial in several ways: the possibility to create virtual worlds with multiple physically separated participants, generating a superior platform for joint projects. Indeed, undertakings in the industrial, corporate and scientific world have noticed the appeal of distributed simulations, as there are a multitude of very diverse areas of application for simulation in general [LAK00]: designing manufacturing systems, analyzing transportation systems or optimizing business processes. With distributed simulation, such projects could accommodate new demands more easily by integrating additional resources. This would in turn enable dynamic scalability and granularity of any developed system. Also, distributed simulation makes redundant porting of models obsolete, since simulations developed in tools by different manufacturers may be hooked together, increasing cost effectiveness while reducing labor time.

The high performance computing community was largely concerned with using this technology to help reduce execution time. This was achieved by subdividing large simulation computations into many sub-computations, and executing these sub-computations concurrently across different processors. In such cases, the principle is being referred to as parallel simulation. The simulations would generally not be geographically distributed, but rather execute in close physical proximity, like on server farms for example [FUJ00]. The military domain on the other hand, was more interested in integrating separate training simulations in order to facilitate interoperability and reusability. The viability of that idea was demonstrated by the SIMNET (SIMulator NETworking) project (1983 to 1995) [MIT95]. This success led to the introduction of a set of rules for the interconnection of simulators denominated the Distributed Interactive Simulation (DIS) standards, as well as the development of the Aggregate Level Simulation Protocol (ALSP) (1990). DIS and ALSP have been supplanted by the High Level Architecture (HLA) protocol since 1996, which now handles interoperability issues for all of the Department of Defense (DoD) simulations [HLA00]. It is apparent that the correct temporal ordering of the logical processes active in distributed simulations represents a major challenge. To this end, several synchronization techniques can be employed. A more detailed description of these will be presented later on however, more specifically in chapter 4.

2.1.2 Distributed Collaboration

Collaboration is the cooperation of two or more participants towards a common goal. The essence of distributed collaboration is to reach this common goal by sharing data, information and actions among the participants. Industrial ventures as well as research projects continue to depend on the expert knowledge of partnered organizations to correctly analyze complex problems and implement suited approaches to solve them. Supported by increasingly sophisticated new technologies, distributed collaboration allows these individuals or cooperating groups to interoperate, regardless of space, time or disciplinary disparities [YAG97], [COM99].

Wainfan and Davis have identified the most notable advantages of the virtual aspect of distributed collaborations as follows [WAD04]:

- *Reach:* The location of any participant is not a factor for a functioning collaboration. Expert knowledge is not an asset required to be on-site.
- *Responsiveness:* Participants have a faster way to interact.
- *Adaptiveness:* Additional participants may be integrated more easily into an existing collaboration.
- *Cost effectiveness:* Heavily reduced travel time and costs.

However, these advantages are also accompanied by less obvious problems. The topic of virtual collaboration in conjunction with information sharing is an intricate subject. Usually, a collaborative project endeavors to reach some form of quantitative or qualitative objective [TUT99]. To achieve this goal, distributed collaboration may involve working with the proprietary information of specific participants, possibly pertaining operating strategies or research details. When secrecy is not an issue, it has proven to be beneficial to fully disclose all information, like in the area of supply chain management [LST00]. But generally speaking, information security is a major concern in the corporate and scientific world, and to most collaborating ventures protecting knowledge assets is a critical prerequisite [FAN06].

The different levels of collaboration and the close relationship to the issue of interoperability are demonstrated in the Levels of Information Systems Interoperability (LISI) Model, developed by the Command, Control, Communications, Computer Intelligence Surveillance Reconnaissance Integration Task Force (C4ISR ITF) in 1998 on behalf of the U.S. DoD [IWG98].

Level	Description	Information Exchange
0. Isolated	No connection	Manual exchange of hard copy data
1. Connected	Electronic connection with separate data and application bases	Homogeneous data exchange
2. Functional	Common functionality with separate data and application bases	Heterogeneous data exchange
3. Domain	Shared data and separate applications	Shared Databases
4. Enterprise	Interactive manipulation with shared data and applications	Cross-domain information and application sharing

TABLE 2.1: LISI Model

The LISI model seen in table 2.1 defines five levels of system-to-system interaction, where each level increases the complexity of the collaboration and thus allows for a higher degree of interaction in between connected systems. But with the gain of interaction possibilities, also comes an increased amount of shared information. The domain and enterprise levels of the LISI model allow for network wide data access. According to the previous paragraph, they are not an optimal consideration for a solution that tries to be applicable to a wide variety of environments, because it would put the knowledge assets of the participants at stake. The approach presented in this contribution therefore focuses on the level two of the LISI Model. The functional level describes systems in networks that allow the passing of complex information amongst themselves. Common functions allow for the sharing of generally heterogeneous data, built from simple formats fused together. The databases however, are kept separately for each member of the network.

2.2 Qualities

After having discussed the fundamentals of distributed simulation and collaboration, it is important to distinguish the many qualities required by the technologies employed in those fields. Therefore, this section will clarify the significance of composability, interoperability, reusability and finally integrability.

2.2.1 Composability

With the functionality to exchange logical data models across programs installed on systems in a network, the possibility to construct modular application-oriented simulations needs to be explored. Kasputis and Henry define "the ability to compose models/modules across a variety of application domains, levels of resolution and time scales" as composability [KAN00]. The defining characteristic of composability according to Petty and Weisel however, is the ability to combine modules and sets of modules at configuration time, in order to build useful and valid simulation systems, tailored to meet a specific set of objectives [PEW03]. This idea allows the components needed for the construction of individual simulation systems to be selected from a repository, as exemplified in figure 2.1.

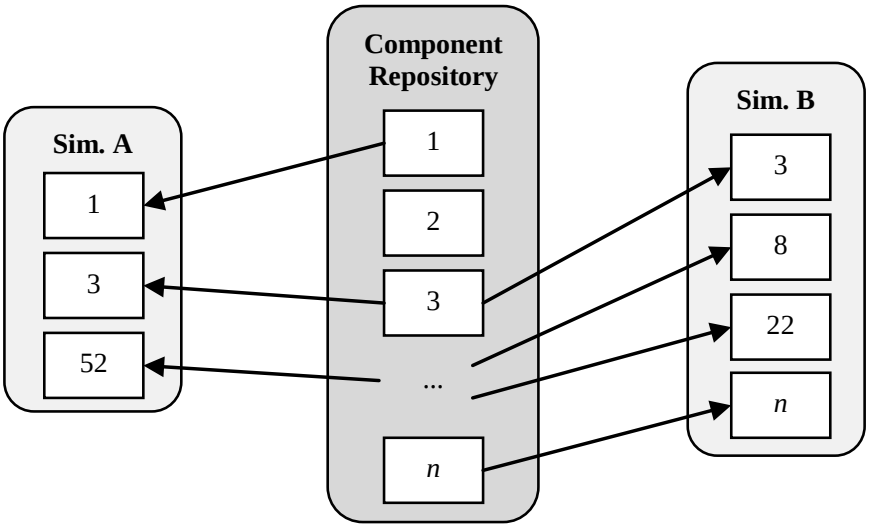


FIGURE 2.1: Composing Simulations (excerpt of [PEW03])

In reference to the unit of composition used during the configuration of a simulation, a total of nine levels of composability have been identified by Petty and Weisel [PEW03]. Due to the modular nature being supported by the approach described in this contribution, the focus will be on the fifth level. At this stage, simulation system components are comprised of individual software executables, which may be federates, federations or even standalone simulations.

A library of such modules must be comprised of valid elements that enable the composition of semantically and syntactically sound systems. In composite simulations consisting of linked models, the input of one component is usually, at least partly, determined by the output of another component. That information flow has to be compatible to produce a valid system. This means that each element has to provide clearly formulated descriptions to identify their functions, explicit specifications of the interfaced inputs and outputs, as well as the intended method of combination. Those rules of operation relate very closely to the idea of interoperability. However, while composability is the ability to combine components at the configuration phase, prior to run-time, interoperability indicates the possibility to exchange data in between components during the execution of a modular system. Therefore it can be said that composability does actually require interoperability.

2.2.2 Interoperability

The Institute of Electrical and Electronics Engineers (IEEE) defines interoperability as "the ability of two or more systems or components to exchange information in a heterogeneous network and use that information" [IEE00]. To successfully identify the level of interoperability of a system, Tolk and Muguira have proposed a layered concept, which they have baptized the Levels of Conceptual Interoperability Model (LCIM) [TOM03]. Their approach is based upon the aforementioned LISI model, and the NC3TA Reference Model for Interoperability (NMI) [CAR04].

A schematic representation of the most recent version of LCIM is shown in table 2.2 [TUR05]. With each level, the degree of interoperability increases:

Level	Interoperability degree	Dimensions
0	None	Integratability
1	Technical	
2	Syntactic	Composability
3	Semantic	
4	Pragmatic	
5	Dynamic	Interoperability
6	Conceptual	

TABLE 2.2: LCI Model

- *Level 0:* Denotes stand-alone systems with no capability to interoperate.
- *Level 1:* Describes systems that have defined protocols enabling them to communicate with each other.
- *Level 2:* Systems interoperating on this level use a common structure to exchange information.
- *Level 3:* Refers to systems that are able to share the meaning of the exchanged data.
- *Level 4:* At this level, mutual awareness of all offered functionalities of the interoperating systems is assumed.
- *Level 5:* Characterizes systems that are able to identify the states and their changes over time of interoperating systems.
- *Level 6:* Requires all systems to be fully specified and documented for interpretation purposes. This implies a system model free of implementational dependencies.

These layers concur with the three dimensions of interoperability proposed by Page et al. [PBT04]. While integrability is the technical possibility to connect multiple systems on a hardware and/or protocol level, interoperability addresses the communication of systems on a software level, and composability is the ability to compose systems of systems on a conceptual level. The distributed simulation approach presented in this

contribution aims at providing the highest degree of interoperability. This means that this new technology shall provide interoperability on the conceptual level according to LCIM, and thusly provide full composability.

Nevertheless, the particular technical interoperability requirements raised by distributed simulation environments need a more in-depth analysis. A distributed simulation is composed of models implemented in specific and generally dissociate modeling and simulation environments. These applications, referred to as Commercial-off-the-shelf (COTS) Simulation Packages (CSP), provide tools to facilitate model design, simulation, experimentation and evaluation. Since interchange formats like the Simulation Data Exchange (SDX) only partially support the translation of models in between CSPs [SLM01], only coupling during runtime is capable of integrating heterogeneous discrete event simulation packages [SLM01]. But the sensible implementation of a distributed simulation consisting of individual CSPs with possibly significantly different modeling approaches and capabilities, is a complex undertaking which calls for a frame of reference on how to actualize interoperability. In an effort to simplify the process, the Simulation Interoperability Standards Organization's (SISO) CSP Interoperability Product Development Group (CSPI PDG) has identified a series of related problem types, represented by Interoperability Reference Models (IRMs). The two major questions that IRMs aim to answer are [TMS07]:

1. How to clearly *identify* the CSP interoperability *capabilities* of an existing distributed simulation.

2. How to clearly *specify* the CSP interoperability *requirements* of a proposed distributed simulation.

To represent these problems effectively, IRMs are defined as basic abstract representations of problem types identified by the CSPI PDG. The use of common modeling artifacts in IRMs allows them to be composable and mapped onto specific CSPs. Ultimately, this makes IRMs a perspicuous tool for the simulation community to understand and interpret interoperability problems.

Type	Description	Sub-type	Description
A	Entity Transfer	1	General Entity Transfer
		2	Bounded Receiving Element
		3	Multiple Input Prioritization
B	Shared Resource	1	General Shared Resource
C	Shared Event	1	General Shared Events
D	Shared Data Structure	1	General Shared Data Structure

TABLE 2.3: IRM Types

Table 2.3 shows the four types of IRMs currently determined by the CSPI PDG [TMS07]. Each type also comes with at least one sub-type. Type A deals with the transfer of entities in between multiple models. The general process is reflected in sub-type 1. Sub-type 2 takes effect when entities can only be transferred if the receiving model is not busy, for example if a process can block the acceptance of new entities. Sub-type 3 describes cases where a model can receive entities from different sources simultaneously, and thus needs to prioritize its input. Type B refers to resources shared across multiple models. Likewise, Type C refers to events shared between different models. Finally, Type D deals with shared data structures that are not defined as resources.

As CSPs are numerous and varied, a heterogeneous distributed simulation involving different CSPs can theoretically present only one or all of the above problem types. In the hopes to satisfy the requirements of a multitude of simulation model combinations, the approach proposed in this contribution plans to provide responsiveness to all four types of IRMs.

2.2.3 Reusability

The approach to reuse software components was introduced by McIlroy in 1968 [MBN68], due to the emerging difficulty to build large reliable software systems. The

proposed idea was to use libraries of reusable components to facilitate the configuration of systems fitted for different fields of application. This technique has now gained a strong foothold in business systems development, on account of lowered production costs, faster delivery systems and increased quality. Sommerville provides a concise summary of the benefits component reuse brings to the table [SOM12]:

- *Costs:* Fewer components need to be specified, designed, implemented and validated.
- *Dependability:* Components which have been already tried and tested, are generally more dependable then newly developed ones.
- *Process risk:* The costs of existing software is known, whereas new developments usually represent a risk factor for a project.
- *Expert Knowledge:* Specialists can create components that encompasses their knowledge.
- *Standards:* Reusability promotes the creation and use of standards. Which makes integration during development easier and acceptability after release more probable.
- *Development:* Reductions in both development and validation tend to speed up system production.

These reasons generally make it more advantageous to develop new systems by defining, implementing and composing loosely coupled, independent components [PCH93]. Such components however, must exhibit certain characteristics to make their usage viable in the construction of composed systems. According to Krueger, abstraction is the most important trait of reusable components [KRU92], i.e. an explicit separation of implementation and interface must exist. On the one hand, a clearly defined interface is necessary for the developer to understand and successfully integrate a component into a new project. Furthermore, implementational changes on a component should not impact other parts of a composed system. Another important feature is that components have to adhere to specified standards to be able to interact with each other; they define

the interfaces and component communication protocols. Additionally, reuse technologies typically provide an integration framework, used by developers to combine several components into a composed system.

In relation to code alterations, two broad reuse strategies can be identified in software engineering: white-box and black-box reuse. Although white-box reuse gives the developers the option to adapt a component to their needs at the implementation level, code modifications also come with complications [RAR03]. The search for components with a suitable functionality in third-party repositories can be very time consuming. A lot of components offer similar functionalities, which must all be examined and understood. This makes a diligent documentation a necessity to allow developers to make an informed choice. This requirement becomes increasingly difficult to fulfill with ever growing catalogs. Black-box reusability tries to mitigate these complications by preventing alterations to the components code. The needed flexibility is instead realized by the possibility to modify various input and output parameters. This eliminates the necessity to understand the inner workings of a component from the users' point of view, thus saving precious development time. On the other hand, the developer of a component does not need to expose the inner workings of his creation. Instead he can choose the type and amount of data exposed, by configuring the interface appropriately.

In respect to the already discussed data privacy issue, this aspect is particularly interesting in the scope of this contribution. Together with the aforementioned benefits, the black-box reusability strategy seems most suitable for a distributed simulation approach, which aims at empowering the component developers to control the amount of shared information, while still providing flexibility and ease of use.

2.2.4 Integrability

Integrability is one further system quality attribute that should be mentioned when investigating relevant aspects of distributed simulation. Integrability is an important concern in the designing process of large, complex systems, particularly in distributed environments. It expresses the ease with which separately developed components of a

system can be made to work together towards a single goal. To achieve this quality, Bass et al. advise to keep component interfaces small, simple and stable, while adhering to clearly defined communication protocols and keeping dependencies between components to a minimum [BCK03]. In a more general sense, Hasselbring explains that the main focus of systems integration lies on three distinct factors: autonomy, heterogeneity and distribution [HAS00]. Integration thus means mitigating the issues that accompany them. To alleviate the effects of distribution, it is advised to use communication protocols that create an illusion of a centralization. The Objects Management Group's CORBA architecture does this for instance, by extending remote procedure calls to remote method calls in an object-oriented setting [OMG12]. Web services work in a similar fashion, by allowing localized access to distant functionality. Heterogeneity is an attribute of system of systems where different hardware platforms, programming languages and data models are employed, so that each individual system may reach its goal in an optimal manner. According to Hasselbring, bridging this factor seems to be the most challenging undertaking of integration [HAS00]. Part of the solution lies with the use of common standards like Extensible Markup Language (XML) to define the shared information [XML08], and wrapping disparate systems in shells that then interoperate trough interfaces, like HLA does for example. Autonomy is a characteristic describing components that have been developed separately, and thus feature distinct interaction and execution concepts.

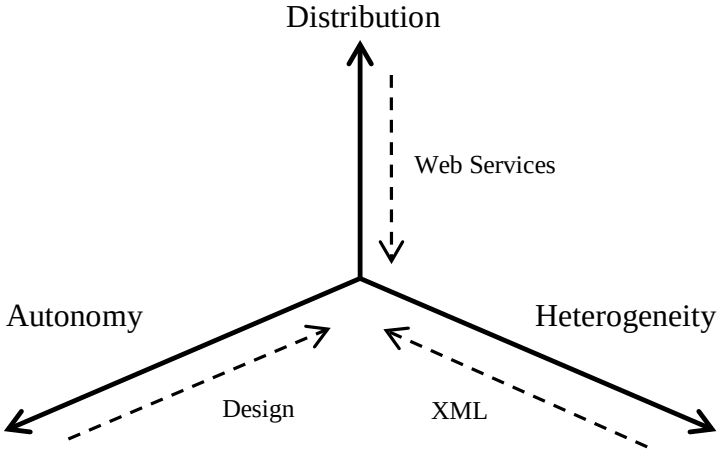


FIGURE 2.2: System Integration Problem Dimensions (confer [HAS00])

Although this makes systems flexible and enables them to quickly adapt to new challenges, it hampers their combination into an integrated environment. Unfortunately, the best course of action to reduce the impact of highly autonomous systems already begins in the design process, so that interaction protocols may be clearly defined. Reducing autonomy afterwards is far too limited. Figure 2.2 expresses schematically the predicament with system integration in a dimensional fashion, with the help of some examples. Moving towards the origin of the system of coordinates attenuates the problems.

2.3 Architectures

Since the goal of this thesis is to present a novel framework to realize distributed simulations, it is relevant to study the many forms software architectures can take. Once the most basic structures have been clarified in the following section, the concept of component based architectures will be addressed. Afterwards, the currently most widespread architecture to create distributed simulations, the HLA, will be presented. In contrast to that, the popular service-oriented architecture will be examined, on which the model pipeline architecture is based upon, which is introduced at the end of this chapter.

2.3.1 Software Architectures

Software architecture design is a discipline that has emerged more recently as software engineers were looking for new ways to create large software systems. As the size of the systems increased, so would the complexity of the overall system structure. It became clear over time that system design specifications represent a more significant issue than the actual implementation. Designing software architectures is the exercise of clearly articulating those design specifications.

An architecture can roughly be understood as the partitioning of a system into parts that relate to each other according to certain principles. That partitioning is what allows disparate individuals or institutions the building of large systems to solve complex problems cooperatively. Each partitioned component then interacts with the rest of

the system through adequate interfaces. Those interaction capabilities are what endows software architectures with their qualities: stability, flexibility, reusability and increased performance to name a few.

Architectural structures provide an abstract framework to understand the concerns of a system in a broader sense; they are key to ascertain the systems abilities and requirements. Over the years, several structure patterns have emerged to better describe certain classes of software. Basically, they describe a repository of components and their mutual interactions capabilities, and define a set of constraints on how those components may be combined. Very commonly used patterns include pipes and filters, object oriented, event based, layers, repositories, interpreters, and process control [SHG96].

Components of an architecture adhering to the pipes and filter pattern, like programs written in the Unix shell [BAC86], have a specific set of inputs and outputs. Each component acts as a filter for an allotted input stream: a set of operations transform the incoming data within the component, which is then written to an output stream, effectively acting as what is more commonly known as processors today. Depending on their position in the pipe structure, these may be designated pre- or post-processors. The processors interoperate with the help of pipes that connect them, which transport data from the output of one component to the input of another. Prevalent specializations of this style include pipelines, which requires the composition of processors to be a linear sequence; bounded pipes, that limit the data capacity of the pipes; and typed pipes, which restricts the data within the pipes to a certain type. Figure 2.3 illustrates the basic principle.

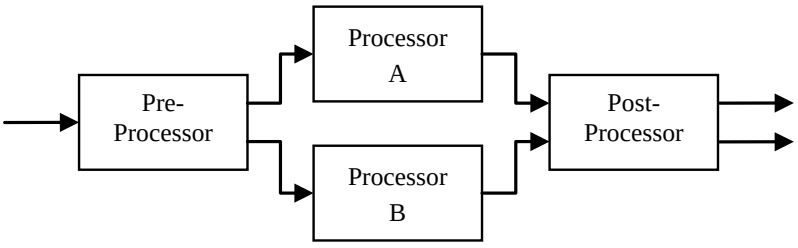


FIGURE 2.3: Pipes And Filters Architecture

Pipes and filter structures have the benefit of clearly conveying the input/output behavior of a system. Furthermore, the simple coupling mechanics promote the reuse of processors. They also pave the way for flexible system enhancements, as new processors may be added or old filters replaced relatively easy. This structure however, generally involves implementing complex transformation operations within the processors, which in turn may hamper the efficiency of the system as a whole during execution.

The widespread object-oriented pattern [KIM89] uses objects as components, which are instances of abstract data types that encapsulate operative data and their associated methods. As each object is held responsible for the integrity of its internal data, the representation thereof is hidden to other objects. They can therefore only interact in a system through explicit invocations of their respective methods. Then again, this property is what leaves interacting objects unaffected by specific implementational details. Figure 2.4 schematically shows how different objects interoperate through procedure calls.

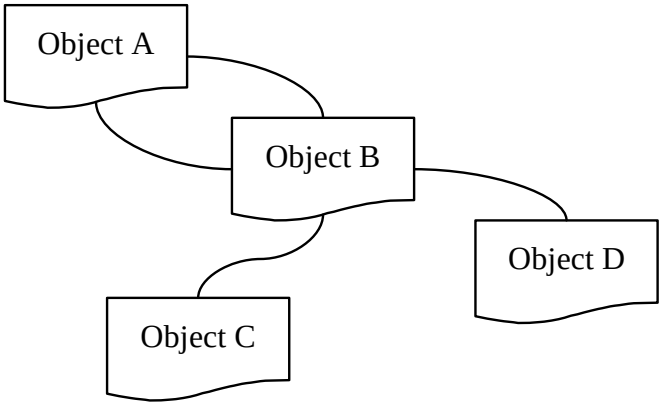


FIGURE 2.4: Object-Oriented Architecture

The downside of object-oriented structures is the need to disclose the identity of objects that need to interact to each other, which is done through importing. This means that in contrast to pipes and filter structures, altering the composition of an object based architecture leads to possibly extensive updates on the participating objects.

The idea behind the event-based pattern, is that contrary to object based structures, components of a system can invoke a procedure implicitly by broadcasting events [GKN92]. Interested components can subscribe to these events, and the system itself calls upon

all associated procedures when they occur. Components of an event based structure are therefore modules whose interfaces provide both access to procedures and a set of events. Figure 2.5 illustrates how different messages prompt an event based system to invoke the appropriate procedures from their modules.

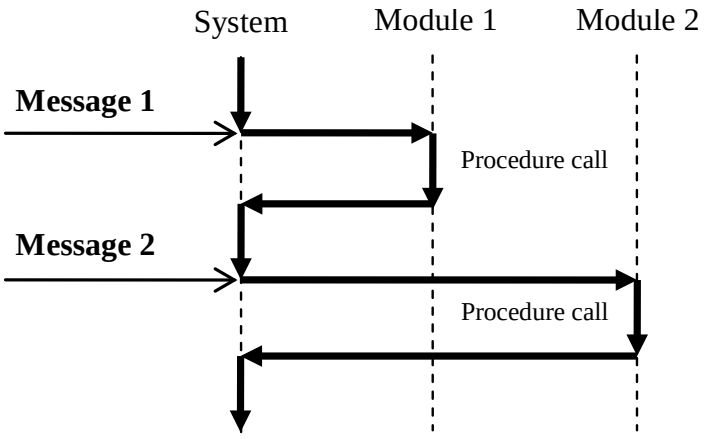


FIGURE 2.5: Event-Based Architecture

This pattern simplifies the introduction of new components to a system, because they only need to subscribe to relevant events. This process leaves the interface of other components unaffected. On the other hand, this approach keeps the components from directly controlling the performed computations. Additional events may be needed by the system for processing acknowledgments, which might be detrimental to the overall performance.

The most widely known examples for layered systems would probably be network communication protocols [COM00]. Structurally, they are a hierarchical composition of layers, interacting with each other through specific interfaces. Each layer serves as a client for the layer below, while providing services to the layer above. This topology allows to partition complex problems into smaller, more manageable ones. One advantage is that alterations to a layered system are rather simple, as it is possible to add, remove or exchange a layer, while affecting at most two others. Another advantage, is that such systems support reuse because of their use of interfaces for their interoperation: one layer can provide services to many others, as long as its interface is supported,

as it is the case with standards like the OSI ISO model [ISO94]. Finding the right levels of abstraction to build a layered system however, can prove to be quite a difficult undertaking. The pattern is illustrated in figure 2.6.

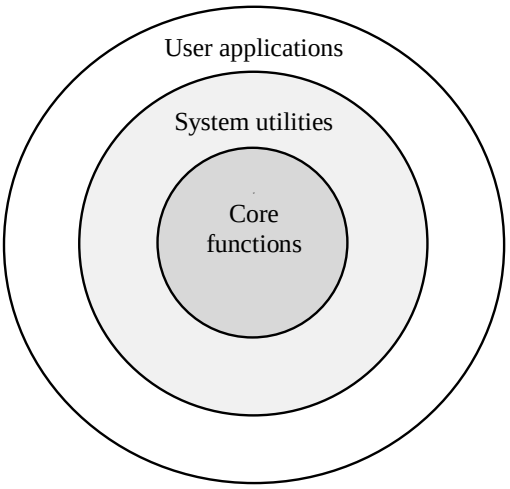


FIGURE 2.6: Layered Architecture

The repository pattern features two distinct types of components: a group of independent knowledge sources interoperating through a central data structure. These knowledge sources make changes to the main data structure over the course of their operation, leading to the solution of the problem the system has been designed to solve. If the state of the data structure triggers executable processes, it is usually referred to as a blackboard. This sub-type of the repository pattern is often the adopted structure for cooperating software agent systems [NLJ96]. Should however the transaction types trigger executable processes, the central data structure generally is some form of database. This pattern is used for instance in knowledge-based systems for information processing [TOH90]. Figure 2.7 illustrates the pattern.

This pattern can be very effective in solving complex distributed computing problems. The scheduling of self-actuating knowledge sources however, can prove to be challenging.

The interpreter pattern is a structure that builds a virtual machine. It usually comes with four components: a program, an interpreter engine, the interpreter state and the program state, as seen in figure 2.8.

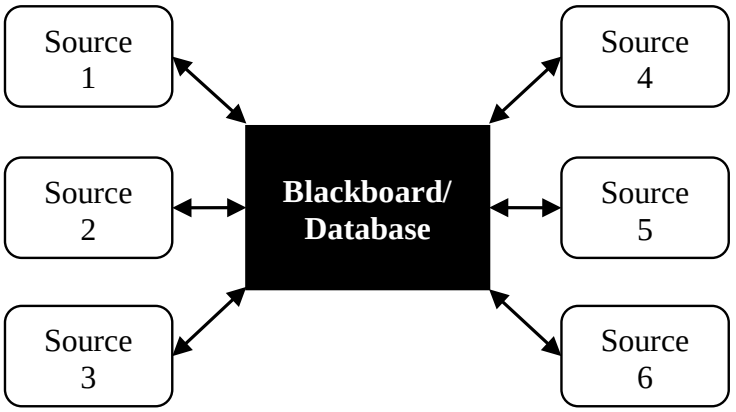


FIGURE 2.7: Repository Architecture

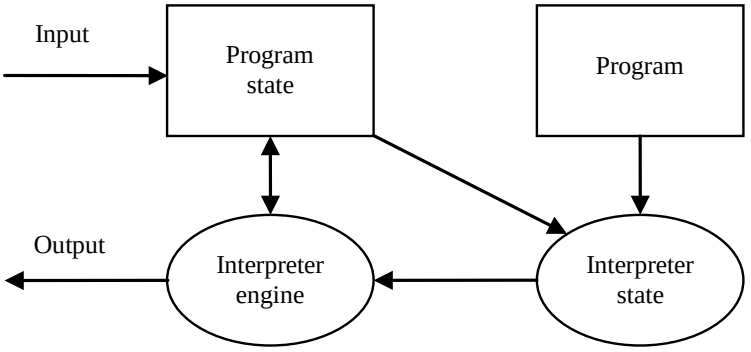


FIGURE 2.8: Interpreter Architecture

The interpreter pattern is comprised of an interpretation engine and the pseudo-code being interpreted. While the pseudo-code includes a program and a record of the execution state of the interpreters version of the program (program state). The interpretation engine in turn consists of the interpreter itself and the current state of its execution (interpreter state). Although being hard to test and design, this pattern is very flexible and highly dynamic because it supports altering the set of capabilities of the system.

Rarely is a complex system entirely homogeneous though. These individual architectural styles are typically combined in a system in one of several ways [SHG96]. In a hierarchical composition, a component of one structural pattern may be represented internally as a completely different one. The filters of a pipe and filter system for example, may be constructed using any other pattern. Another way to combine structures, is to enable a components interface to operate in multiple ways. For instance can a component act as a knowledge source for a repository, while interacting through pipes with

other components.

Since modern simulators are among the most complex software systems created and maintained in existence [BCK03], it is therefore no surprise that the intelligent design of software architectures is an important field of research in the world of simulation.

2.3.2 Component Based Architecture

Component based architectures enable the design of simulations based upon encapsulated models communicating with each other to exchange data and provide global access to otherwise not available functionality. This concept was spearheaded by people such as Zeigler [ZEI87], who exposed the advantages of the modular coupling of models.

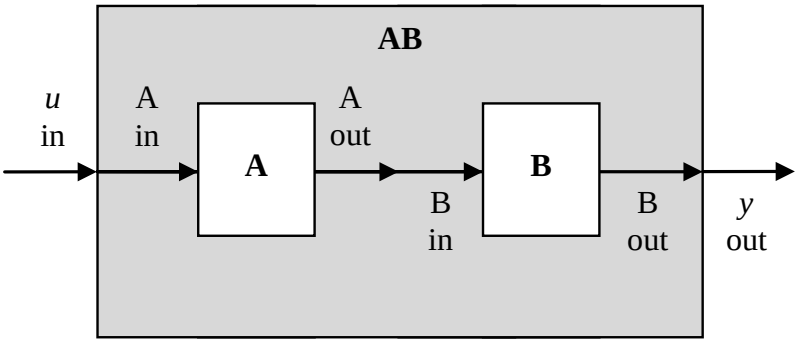


FIGURE 2.9: Modular Composition (confer [ZEI87])

The "coupling" of models is a term first introduced to the world of simulation in the 1980's by scientists such as Ören [OEC80]. Figure 2.9 shows such a coupled model AB, composed of two atomic models A and B. Zeigler demonstrates in this manner, how coupling can generally be achieved between arbitrary models, if they describe their in- and outputs in an interfaced fashion. As can be seen, an input u can result in the output y by applying the transformation $y = (AB)u$. This modularization, in turn, enables the hierarchical construction of models, since this basic principle may be applied over and over again to create constantly growing models.

Figure 2.10 describes the general pattern with which one may construct hierarchical models. An atomic model is a singular model that is not coupled to other models. A composite model on the other hand, is a model comprised of atomic and/or other

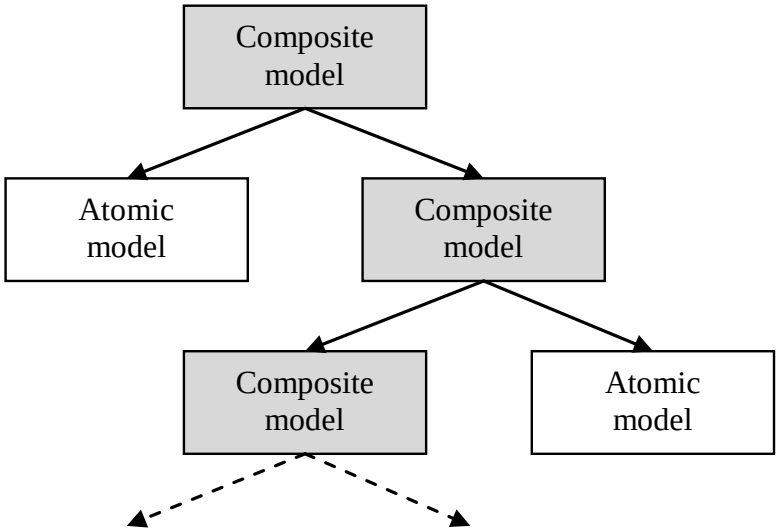


FIGURE 2.10: Hierarchical Composition

composite models. Those models may in turn be coupled atomic and / or composite models. This pattern may be reiterated to any arbitrary depth, to create increasingly complex composite models.

The benefits of such an approach to model building are obvious. One important aspect is the ease to efficiently verify more complex models: a system can be dissected into its core processes, which in turn can be implemented as atomic models. Those can then easily be tested in early development stages, simplifying the verification process of the composite model of the original system [ZEI87]. Another significant factor which has already been examined more closely, is the reusability of well tested models for the development of new complex models.

The question remains however, on how to define a component when talking about the subject of discrete simulation. Pidd identified various requirements, which an application has to fulfill, to be viable for use in a composite model [POB99]:

- *Encapsulation:* The access to bundled data and methods of the application has to be restricted to ensure consistent definition of its functionality.
- *Interfacing:* A component has to feature an interface to allow communication with other components via message passing.

- *Explicitness:* The interface has to be accurately defined, to describe the accepted input and produced output of the component, which also includes error handling capabilities.
- *Accessibility:* All functionality of a component has to be accessible from other components in a system.
- *Conformity:* All components of a system have to abide by the defined coupling rules of this system.

There are various component based architectures for discrete simulation in existence. Approaches such as the Discrete Event Systems Specification (DEVS) [ZSK95], the JavaBeans Discrete Simulation (JBDS) [FYW98] or the Visual Simulation Environment (VSE) [BBE97] provide platforms for the creation and execution of composite models designed to meet the above criteria. The popular HLA framework is especially relevant in this case, as it takes existing, independently executable models and wraps them inside an object-oriented shell to enable the use as component in component based simulations.

2.3.3 High Level Architecture

In 1996, the baseline HLA definition "HLA 1.0" was approved as the standard technical architecture for all DoD simulations [DFW97]. In the following years, the first implementations of the interface specification and the HLA Runtime Infrastructure (RTI) ensued. In the year 2000, the version 1.3 of the HLA was accepted as an international standard in an almost unchanged form by the IEEE [HLA00]. The use of HLA in the civilian simulation community has been spearheaded by academia, where it was mostly involved in research projects [SSK99], like the HLA simulator for land based transportation [MOP06].

The HLA is an architecture that guides the building of systems, with a clear set of rules to direct the interaction between their composing elements. It features the characteristics of three structural styles: it is a layered system of object-oriented components interoperating through implicit invocation. The RTI operates as a layer, separating all generic interoperability services from the individual simulation models. This avoids to

implement redundant code within them and allows the RTI technology to be updated without needing to alter any of the other components. These components in turn, are treated as objects by the RTI. They may be single atomic simulation models or encapsulate a whole other HLA simulation; as long as their interface adheres to the HLA principles, the RTI can work with them. Lastly, none of the HLA constituent simulations directly interact with each other, the RTI is always involved. As specified by an event-based architecture, the composing simulations have to subscribe to certain events to be able to act on them. Creating an event thusly means implicitly calling upon the services of other participating components of the simulation. It is therefore not necessary in theory that the composing simulations know of each another. In practice however, semantically coherent simulations obviously still require some level of mutual awareness during the design process.

In accordance to HLA specifications, a distributed simulation, called a federation, essentially consists of four components: the RTI, a Simulation Object Model (SOM), a Federation Object Model (FOM) and a set of modular applications, referred to as federates. Generally, federates are atomic or composed simulations, but may also represent data viewers or collectors, as long as they incorporate the ability to exchange data with the RTI. The notion of federates comes from the idea of several states joining together as a group, a federation, while still retaining a certain personal autonomy. A FOM is a common structure for the data exchanged in between federates of a specific federation. A SOM is the documentation of all the simulation functionalities of the federation. The RTI functions as a control unit, offering the federates services established by the HLA interface specification. The RTI also represents the central communication hub, allowing all the connected federates to exchange information. It is not tied to a specific implementation, and can therefore take a variety of forms. A schematic representation of an HLA federation is shown in figure 2.11. The functionality of the RTI is described more closely in a working example of an HLA federation in chapter 6.

The HLA standard is defined by three essential parts which will be discussed in the following [KWD99]. The HLA rules are a set of conventions that dictate the Interface

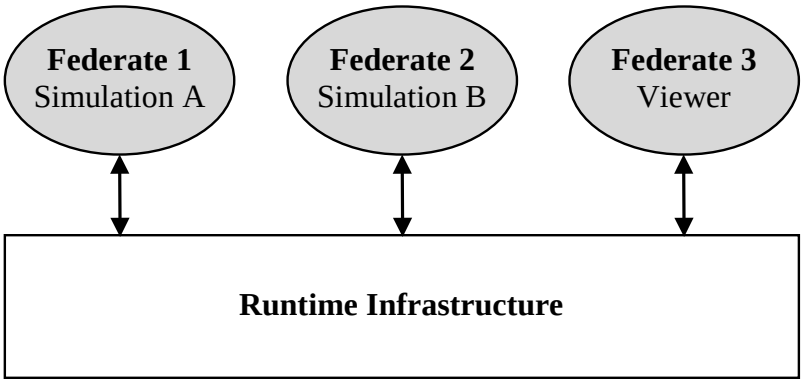


FIGURE 2.11: HLA Federation Overview

Specification (IS) and Object Model Template (OMT) design. Furthermore, they assure proper interaction of federates during the execution of a federation by formulating behavioral principles for both.

Federation rules:

- All federations have to define a FOM which adheres to the specifications of the OMT.
- All instances of objects declared in the FOM are located in the federates, not the RTI.
- All the data exchange between federates is conducted by the RTI.
- All federates interact only according to the IS with the help of the RTI.
- During a federate execution, only one federate can hold the ownership of an attribute declared in the FOM at any given time.

Federate rules:

- All federates have to define a SOM which adheres to the specifications of the OMT.
- All federates must be able to operate according to the attributes and interactions declared in the SOM
- All federates must have the ability to dynamically take or relinquish ownership of an attribute defined in the SOM.

- All federates must be able to react to the conditions defined in the SOM, and handle attribute updates accordingly.
- All federates must manage their internal clock in way that lets them coordinate with other federates of the same federation.

The IS defines the interface the RTI presents to federates and vice versa. It is an abstract construct that regulates the interaction between federates and the RTI by establishing a design standard for the offered services of their mutual interfaces. These services fall into six possible categories:

- *Federation management:*
  Comprised of federation wide operations, e.g. services to create a federation execution, control over joining or resigning of federates and coordination of federation checkpoints.
- *Declaration management:*
  Federates do not send data to other federates by name. In fact they publish and subscribe to data produced in the federation with the help of services. The RTI then uses this information to manage the data flow.
- *Object management:*
  Services of this category are used for object manipulation within a federation, e.g. register new objects or update attributes of an existing one.
- *Ownership management:*
  Represents services that facilitate ownership acquisition or transfer of an object instance by a federate for attribute updates.
- *Time management:*
  expressed with the abstract notion of logical time. The services of this category allow for the coordinated advancement of logical time within a federation, and take care of the delivery of time-stamped events.
- *Data distribution management:*
  Provides services dedicated to the abstract routing of object instance information, i.e. filtering data to reduce irrelevant information from communication between federates.

The OMT acts as a meta-model for all FOMs and SOMs, and as such is key in defining their structure. Essentially, they represent a record of the modeling artifacts used in a specific federation:

- *Object class structure:*
  A table that lists the namespace of all federation or federate object classes and the relationships between their subclasses, possibly in a hierarchical fashion.
- *Interaction class structure:*
  A table that lists all possible interactions among objects of a federation or federate.
- *Attribute/Parameter:*
  A table that describes in detail the characteristics of object attributes and interaction parameters of a federation or federate.
- *Data type:*
  A table that specifies the data representation in the object model of a federation or federate.
- *FOM/SOM lexicon:*
  A table that documents all the terms of the previous tables, and explains them verbally to ensure a semantically correct use of them.

It is apparent that in the HLA there is a strict separation of syntactic and semantic interoperability. While it does provide the necessary syntax for interoperability, the HLA presents the users with a framework that allows them to define the semantics of their own simulation by creating the appropriate object models in the FOMs and SOMs. The process of developing these is supported by different tools, like Pitch Visual OMT [PIT10].

The HLA has progressed rather rapidly from a concept to a standard in a period of less than four years, mainly due to the backing of such a strong institutional proponent such as the DoD. Since it used to be a requirement for the HLA to be used across all DoD simulations in the initial phase, the technology has gained enough users to be adopted successfully. This has led to a wide range of tools facilitating the development of HLA enabled simulations, due to the active user base. Still, authors such as Boer

and Strassburger postulate the thesis that the HLA is far from becoming a mainstream standard for civilian simulations [BOE05], [STR06]. It is an often heard opinion that using the HLA is too complex, too error-prone, due to its federate interface specification [STR06], [HHH12]. In this context, it has also been noted that the performance penalties are significant and that the overhead it causes can be problematic [STR06], [DAM99]. Furthermore, the growing importance of net-centric simulations has uncovered many deficiencies on interoperability, extensibility and reusability of the HLA [NHT04]. As an IEEE standard, the HLA needs to be reviewed every five years. And so it came that the HLA Evolved Web Service API was developed as an enhancement of the standard, during a revision in 2006 by the SISO HLA Evolved PDG, to face these issues [MOD06], [MOL06]. Although this constitutes an important advancement of the technology, many problems of a service-oriented HLA still remain unsolved, such as sensibly defining interactions between the fine-grained HLA services and their less detailed web service counterparts [WYL08].

2.3.4 Service-Oriented Architecture

As Papazoglou and van den Heuvel justly put it, services constitute the next major step in distributed computing [PAH03]. They are interesting for several stakeholders. To software engineers for example, they are a means to create dynamic and collaborative applications. To IT managers, they allow for the effective integration of the usually rather diverse enterprise systems. For business managers on the other hand, service orientation brings information technology investments more in line with business strategies.

Service-Oriented Architecture (SOA) is a framework proposed by the Gartner Group in 1996 [SCN96], which envisages individual units of logic, called services, to work autonomously, while adhering to a common set of principles. The key aspects of these principles are as follows [LJD01]:

- *Loose coupling:* Services only require to be made aware of each other through service descriptions to minimize dependencies.

- *Service contract:* The service description defines communication rules that the service always has to adhere too.
- *Autonomy:* Services have exclusive control over the logic they encapsulate.
- *Abstraction:* Services hide the logic they encapsulate, except for what is specified in their service contract.
- *Reusability:* Reuse is encouraged by partitioning complex logic into simpler services.
- *Composability:* Services promote their assembly into more complex composite services.
- *Statelessness:* Services do not persist their state in between activations.
- *Discoverability:* the service description is designed to be findable and accessible through the appropriate mechanisms.

Services encapsulate logic in a specific context to retain their independence. Furthermore, they may be comprised of logic provided by other services. This allows to represent a process as a service, encompassing other services that stand for the steps needed to execute that process, as seen in figure 2.12.

A very suitable platform to build service-oriented solutions are Web Services (WS) [WSA09]. It is important to clarify that the architecture and the technology are not interchangeable: various SOA models existed before the introduction of WS. But WS are a very successful standard that has proven to be a proficient implementation of SOA [LJD01].

WS originated from the emerging need of organizations to exchange information between themselves through the Web in the 90s [SOM12]. Since access through web browsers or the development of single purpose communication programs is generally impractical, the idea of a standardized web-based distribution framework arose. The concept called WS allows to encapsulate the often highly disparate data structures of different organizations into standardized shells, which allows to offer information as services to a wide range of users [TBB09]. The public interface of a WS is a central component that assigns the WS its identity. This interface is built using the Web

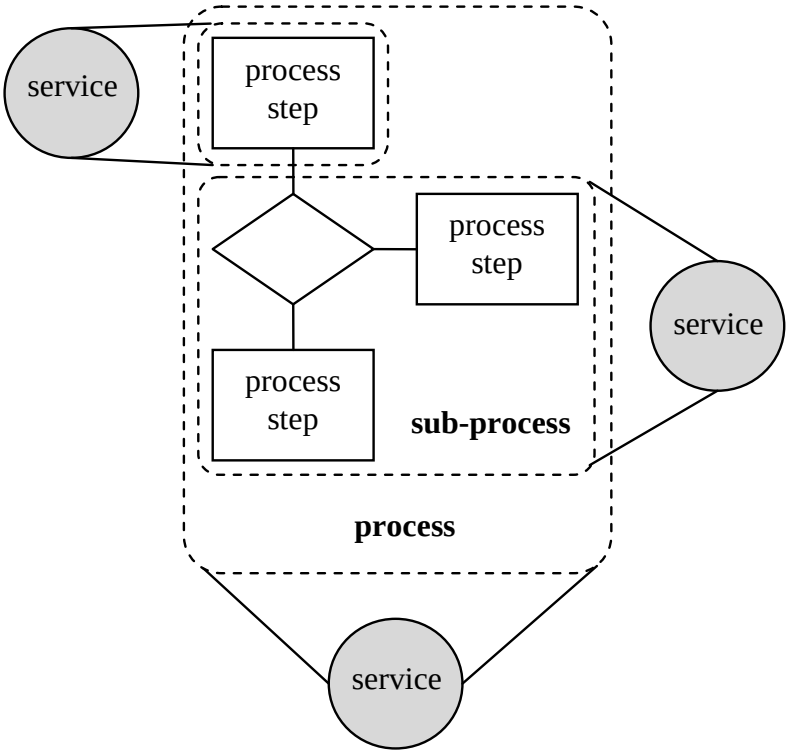


FIGURE 2.12: Logic Encapsulation with Services (confer [LJD01])

Service Description Language (WSDL), which was submitted to the World Wide Web Consortium (W3C) in 2001 [WSD01]. The data exchange between WS is usually done using a standardized messaging protocol, originally defined as Simple Object Access Protocol (SOAP) [SOA07], although alternatives such as XML remote procedure call (XML-RPC) exist [LJD01]. Optionally, publication and discovery of WS is supported by the Universal Description Discovery and Integration (UDDI) registry, which was developed by UDDI.org and the Organization for the Advancement of Structured Information Standards (OASIS) [OAS02], but has been superseded in recent years by regular web search engines [SOM12]. All of these protocols, WSDL, SOAP, XML-RPC and UDDI, are based upon the XML standard, which was created by the W3C as a derivative of the Standard Generalized Markup Language (SGML) [SGM95]. XML is used to establish the format and structure of WS messages. Several additional specifications emerged with XML as a base, which are of great use to the WS standard. The XML Schema Definition (XSD) language allows to build and confirm valid XML document data [XML00]. And the eXtensible Stylesheet Language Transformation (XSLT) standard is used to transform XML information from one schema to another [XSL99].

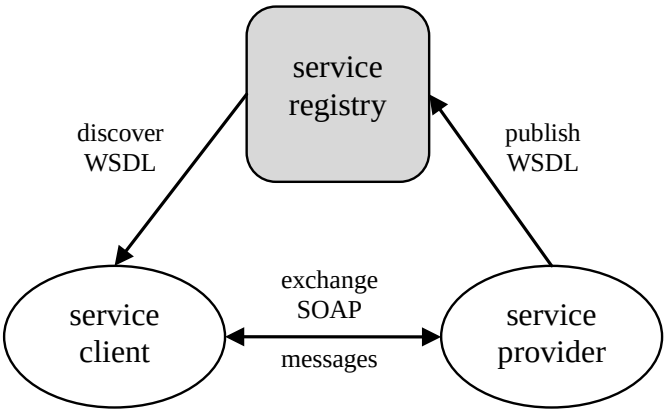


FIGURE 2.13: Basic Components of SOA (confer [LJD01])

Figure 2.13 is a representation of how the basic WS standards interact to create a SOA. A service provider implements services and specifies their interface with the help of WSDL. The interfaces are then published in a, possibly UDDI conform, service registry. A service client is now able to discover these services through their WSDL interface in the registry, and locate the provider. The client application can then be bound to a specific service, and communicate with it through SOAP messaging.

The primary characteristics of SOA have been explored by Erl [LJD01], and shall be summarized here to emphasize the usefulness of the paradigm in distributed modeling and simulation (M&S) approaches:

- *Increased quality of service:*
  Tasks can be carried out in a secure and reliable manner. What the interface of a WS exposes, as well as the messaging content, is entirely specifiable and is inherently protected by the WS standards. Furthermore, message integrity and delivery, or notification of failed delivery, can be guaranteed.
- *Fundamental autonomy:*
  As previously mentioned, services of a SOA are inherently independent, because of the requirement to maintain control over their encapsulated logic.
- *Use of open standards:*
  The use of standards like XML, XSD, WSDL and SOAP allow the exchanged messages to be fully self-contained, due to the data representation only requiring to be specified by the WS description.

- *Intrinsic interoperability:*

  WS naturally promote interoperability with the requirement to use open standards. This can significantly reduce the cost and effort to achieve cross-application integration.
- *Federation promotion:*

  WS do not require to replace any application logic. On the contrary, SOA serves to encapsulate and expose it via open communication standards, which promotes unity across non-federated environments.
- *Architectural composability:*

  As has already been pointed out, services exist as independent units of logic. And a process can be broken down into a series of cooperating services, demonstrating how a composition of different services can work together.
- *Inherent reusability:*

  Services naturally promote reuse as a side effect, since they encapsulate a discrete functionality, which may be distributed and programmatically accessed in a variety of projects.
- *Immanent extensibility:*

  SOA oriented solutions can be supplemented rather easily with services adding extra functionality, while having only minimal impact on their published interfaces.
- *Abstractional layering:*

  WS represent access points to a variety of resources and application logic. All the implementational details and proprietary data formats involved can be hidden.
- *Increased agility:*

  WS remain largely unaffected by changes in the implementation of an application, or the replacement of an established technology, due to the use of service layers and interfaces, and loose coupling.

Due to these measurable and tangible benefits, several M&S architectures have emerged in the wake of SOA. The now discontinued Extensible Modeling and Simulation Framework (XMSF) and the still relevant Cosim-grid have been paving the way as a new generation of web-enabled simulation technologies [TFC06]. XMSF attempts to improve

the interoperability of M&S applications with XML and Web services in a networking environment. In this context, XMSF implements a web-based interface for the HLA RTI, allowing the access through established Internet communication standards such as SOAP [BZP02]. Also based on HLA, is the service-oriented simulation grid Cosim-Grid [LCD05]. Its aim is to improve upon HLA by adding mechanisms for dynamical sharing, increased autonomy, fault tolerance collaboration and security. Model pipelines have also been implemented using a service-oriented approach by adopting WS and their auxiliary standards. The framework does however not support the HLA paradigm, but instead tries to create a more accessible solution for the coupling of simulations in heterogeneous environments.

2.3.5 Model Pipeline Architecture

Model pipelines are a novel approach to realize distributed simulation, which tries to emphasize on the collaborative aspects of the subject. This means that the coupling of disparate simulators in different geographical locations represent a major focus, which is not the case in classical coupling mechanisms [WHM09]. This allows for a much more flexible employment of simulation environments, one that is best suited for the models needed in a joint project. This also allows already existing models to be integrated without porting their implementation to a new environment. Finally, possible issues concerning costs of procuring additional licenses for CSPs are remediated.

The foundation for the model pipeline concept has been laid out by Bach et al., while working on an approach to create an e-learning application capable of granting web-based access to the "Oxsoft Heart " simulation [BHW03]. To that effect, the simulation has been embedded in a client-server architecture. This creates an intermediate layer, separating the model description on the server, from the input specifications and output representation on the client side. This abstraction makes it possible to personalize or restrict the user's view of the simulation. It would also allow to design standardized user interfaces for simulations created in different simulation packages, removing the need to familiarize oneself with a constantly changing working environment. Furthermore, since the simulation and its runtime operate as a black-box behind an interface, only

limited knowledge of the actual model implementation is necessary to be integratable in a server environment.

Wittmann et al. have then postulated that, within the scope of a project with multiple collaborating partners, such a configuration could be beneficial [WHM09]. Several simulations created with different simulation packages, operating on their respective servers, could interact given the addition of a component containing the necessary information to describe how to link the individual models. But different models usually also greatly differ in their adopted paradigms, degree of detail, time resolutions and overall implementation. Instead of going for a complex scaling approach like what the HLA offers, Wittmann et al. have proposed a more straight forward solution, in the hopes of simplifying the coupling process: a one-directional model chain where the information is passed through pipelines.

The basic idea to couple processes trough pipelines goes back to McIlroy [MCI64] and became more popular as Thompson implemented them as "pipes" in the Unix operating system [LIP06]. The principle stipulates that the output stream of a preceding process is to be used as input stream for a successor process. More than two processes may be involved in this chain, allowing a process P2 to act simultaneously as successor to a process P1 and predecessor to a Process P3.

However, the proprietary character of the above client-server concept interferes with the idea of a simple implementation of an openly accessible model chain. Therefore, the realization of that idea as SOA was proposed [WHM09]. Instead of proprietary implementations of servers to interface with the different models, they are to be published as WS, by wrapping them in a service-oriented shell. Information exchange in between WS can then be handled according to the SOAP messaging standard. The already mentioned component defining the model linkage can be implemented using the XML Pipeline Language (XPL). This XML-based language has been submitted to the W3C in 2005 by Bruchez and Vernet [BRV05], and creates a framework for the comfortable creation of pipeline structures. It constitutes a core component of the Orbeon Forms Platform [ORB11], which is an open source solution for the creation of XForms based web-interfaces [XFO09].

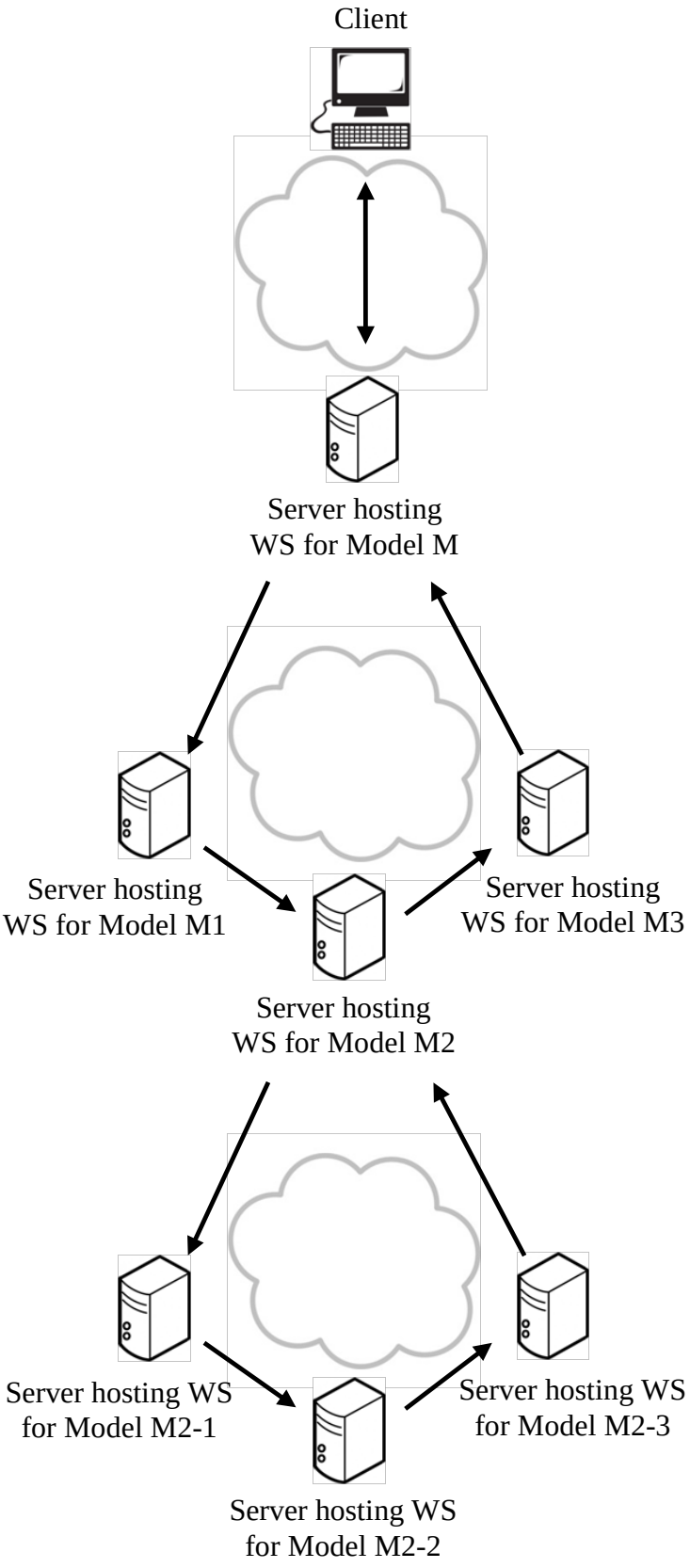


FIGURE 2.14: Accessing a Model Pipeline

Although the model pipeline approach in itself is not reliant on specific technologies, Orbeon Forms presents a convenient way to implement model pipelines through its multitude of ready-made XML pipelining language (XPL) processors. They facilitate many basic tasks like the integration of XML or Structured Query Language (SQL) [SLA04] databases, XSLT transformations of XML data, the invocation of JAVA programs [GOS00], and the delegation of data to the applicational logic of external services, like WS. Furthermore, Orbeon Forms applications do not only have the ability to consume WS, but can be created and exposed as such, through the use of XPL. Therefore, the more complex structure of a hierarchical composition of model pipelines could be envisaged.

A system designed with the technologies proposed here, would look like what is shown schematically in figure 2.14. Given the proper authorization, a client could access a model M, hosted on a server with a WS as interface. This model is a chain of the models M1, M2 and M3, which in turn, are hosted as WS on different servers, linked together by a model pipeline. To illustrate the hierarchical capabilities of such an architecture, in this example, the model M2 is actually another model pipeline: it is composed by the models M2-1, M2-2 and M2-3, which also have each a WS interface, located on distinct servers. So the client is actually working with a distributed simulation, composed by five models, coupled by model pipelines.

From an architectural stand point, this approach is a heterogeneous composition of different styles, as it is often the case. The main style here, is obviously a service-oriented one. The different models of a distributed simulation realized with model pipelines are treated as stand-alone, black-box systems. Given a proper interface description, they are then exposed as WS to be executed on geographically distributed computers. However, the model linkage very much follows the style of a pipes and filter architecture, which is analogous to the already mentioned Unix pipes. Using figure 2.14 as reference, model M1 is computing data from the simulation parameters originally sent by the client to M. The output stream resulting from M1's execution is then communicated through its WS interface as SOAP message to the WS representing M2. M2 in turn, is then using that information for its own execution to calculate an appropriate output. A similar process then occurs between M2 and M3. Finally, model pipelines also show characteristics of

a layered architecture. The WS acts as a client to its model through a specific interface. This allows to abstract the implementational details from the service requestors, and enables standard communication protocols in between WS.

Chapter 3

Simulating The Processes in and around an Airport

3.1 The Airport2030 Project

The first working model pipeline prototype has been implemented within the context of the Airport2030 project, which is introduced in this section.

Because of the constant growth of the air traffic sector in Germany [IAT11], the Federal Ministry of Education and Research (FMER) commissioned the Airport2030 cluster of excellence project, to uncover available optimization potential for the time horizon of 2030. According to the project board, an increase in air traffic of 300% is anticipated within the next 20 years [AIR11]. For this reason, it has become an important objective for concerned enterprises and research fields to reduce transport costs, emissions and noise factors, while improving upon the handling times and punctuality.

With the Airport Hamburg as example, the Airport2030 project wants to develop and showcase new technological approaches to enhance the execution of ground processes, which affect the overall quality and performance as well as the environmental impact of the air transport system. Indeed, the passenger and luggage handling related processes inside the airport terminal, supplying, loading and unloading planes at the gates, as

well as the traffic generated on the apron in this manner all show great potential for improvement. Studies conducted by Airbus have uncovered that with optimized ground handling processes, it would be possible for an airline to increase profit by 25% by having the possibility to offer more flights while maintaining their current fleet size [AIR11]. In addition to the economical factor, efficient ground handling processes also have the potential to decrease emissions of vehicles on the apron to great effect.

Several partners from universities, research organizations and industry have come together to focus their expertise in the fields of engineering, logistics, air traffic management, system development and simulation, to discover new insights affecting airport operating procedures and their antecedent and subsequent processes to potentially optimize them. These essentially involve everything concerning the passenger flow from their respective homes to the airport, their navigation trough the terminals, as well as boarding procedures and all necessary activities to handle the planes. This holistic approach takes into account the interaction of the networked components of the air transport system. In light of this project, the airport is therefore split into distinct subsystems, as seen in figure 3.1.

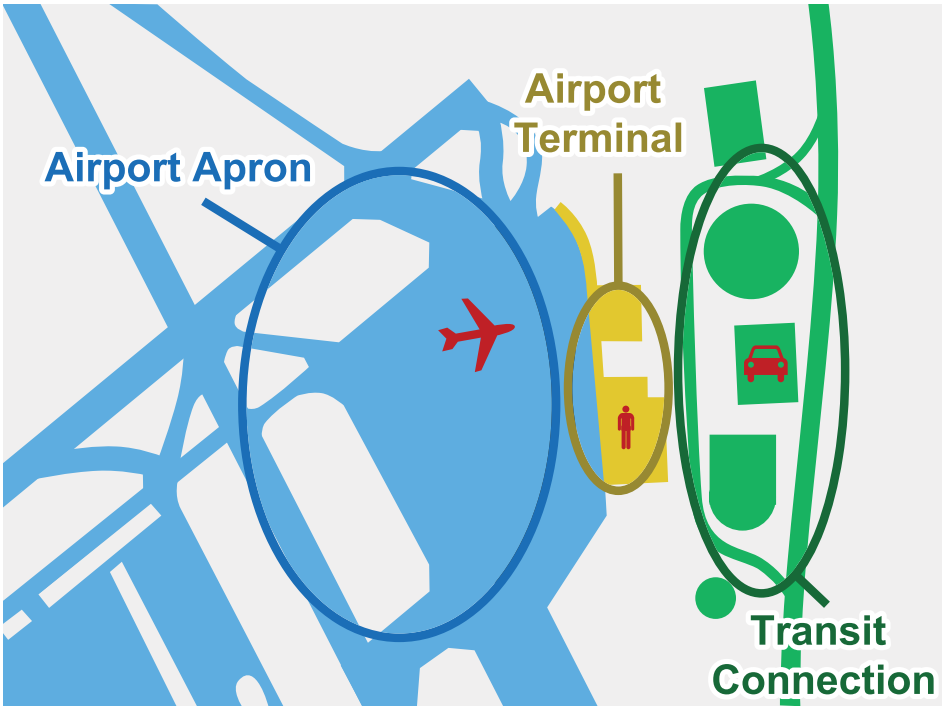


FIGURE 3.1: Subsystems of an Airport (confer [AIR09])

This arrangement is the basis for the structuration of the Airport2030 project and serves as fundament for the development of the necessary models to analyze the current state of the airport process flow. The overarching goal of this project is to create and evaluate different scenarios with these models, representing possible outcomes of new organizational and technical developments for the year 2030. The aforementioned partners have the task to generate models for the transit connection, airport terminal and airport apron subsystems, and more relevantly, coupling them into a composed simulation to enable an adequate overall analysis.

3.2 Processes of Interest

Now that the project has been outlined, it is necessary to identify the relevant processes that need to be abstracted to build an expressive simulation. It is relevant to note that the project specifically focused on the processes of outgoing flight procedures. Because these proved more complex and time consuming than their counterparts during the arrival of planes, the optimization potential was estimated to be more promising. Figure 3.2 shows a basic breakdown of the main processes taking place at an airport for departing flights.

3.2.1 Transit Connection

The Technical University of Hamburg-Harburg (TUHH) as the workgroup dealing with the transit connection to the airport, has created a multi-modal traffic generation model for the Airport2030 project. Using synthetic population data, this model aims at representing the whole of the local traffic of the metropolitan area of Hamburg. This allows it to simulate the flow of private and public transportation from passengers from their doorstep to the Airport Hamburg [LBB11]. Each passenger is represented by a set of data mainly holding information about:

- Their age and sex.

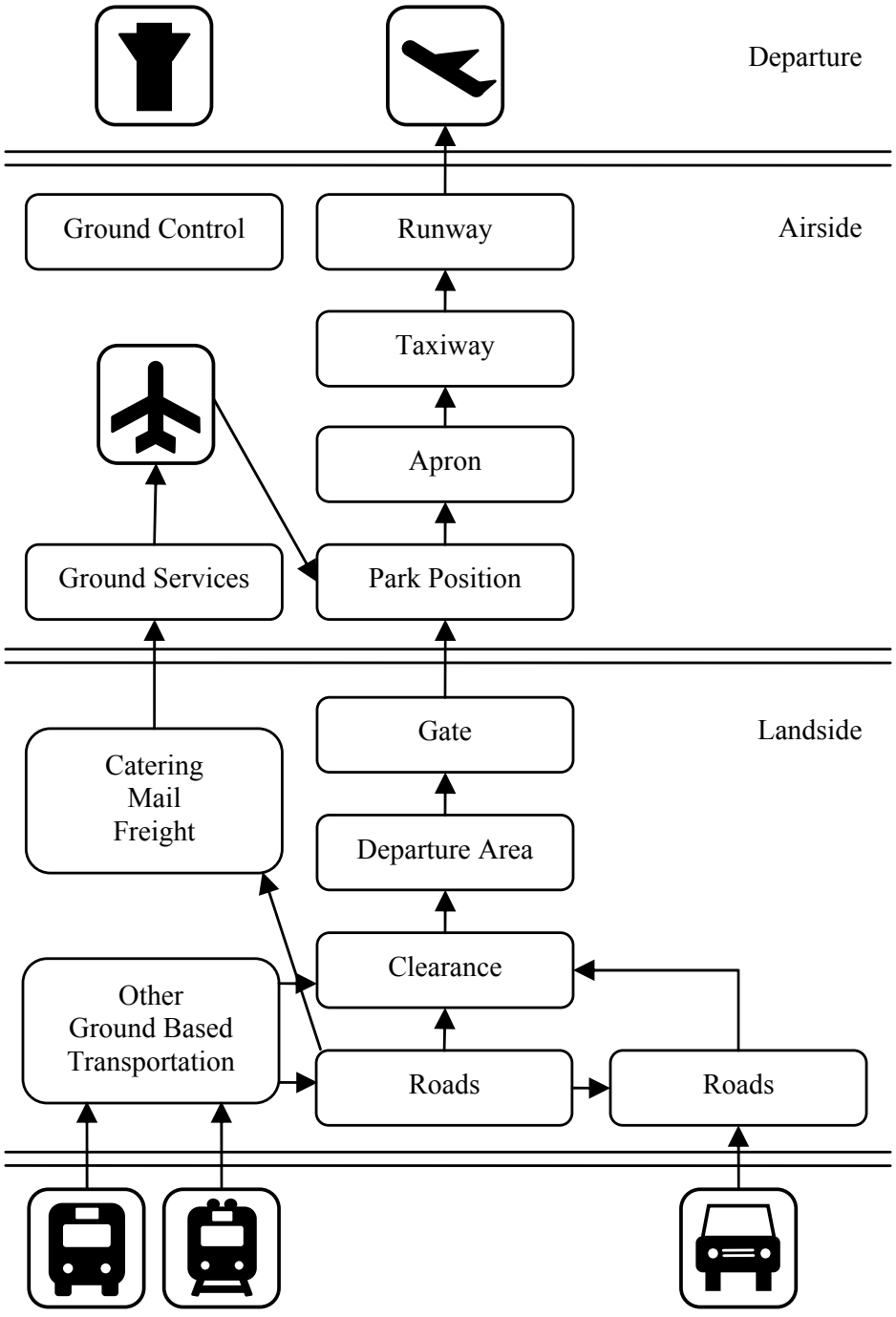


FIGURE 3.2: Departure Processes at an Airport (confer [DAE02])

- Their travel reason (tourist/business).
- The flight they are boarding.
- The aircraft they are boarding.
- The time of arrival at the airport.
- The time of departure of their flight.
- The location they started their voyage.
- The location they are flying to.
- If they are traveling with luggage.

The goal here is to uncover changes in transport demands and how to cater to them [LOA11]. Three scenarios were created with the help of an intuitive scenario approach based on an extension of existing ACARE scenarios [ACA04]. The first scenario simulates an inhibited economic growth, the second the rise of more efficient technologies, while the third anticipates increased attention to security measures. More information on the specifics of these scenarios can be found in the scenario creation description of the Airport2030 project in [PHL09].

3.2.2 Airport Terminal

The University of Hamburg (UHH) is responsible for modeling the passenger and luggage flow through the airport terminals. Each terminal has been modeled with different possibilities of checking in, security checks, shopping options and use statistical data to reproduce the travel time between those stations. Each of these stations is represented adequately with the necessary metrics like quantity or operational delay [FFH10], [WWH10]. The complete workflow can be summarized as follows:

1. Passengers reach an airport terminal by car, taxi, bus or subway. These numbers are simulated by the Transit Connection model.

2. These passengers then travel from the entrance to the check-in counters. The model differentiates between manned check-in counters, automated check-in machines, and online check-in.

3. After receiving their boarding pass and turning in their luggage if any was present, the passengers travel to the security check stations.

4. After having passed the security check successfully, the passengers travel to their respective gates.

5. If there is time remaining until boarding, passengers may use the time to take advantage of the shopping opportunities, or just wait in the appropriate areas.

6. During this time, the baggage items are moved from the check-in counters to a collecting point, ready to be transported to their assigned planes.

7. Lastly, the passengers go through the boarding process, which includes a check of the boarding pass and possibly their passport.

It serves the study of the organization of terminal processes and the analysis of new indoor navigation technology.

3.2.3 Airport Apron

The airport apron subsystem has been modeled by the German Aerospace Center (DLR). It uses flight plan data to feature stations to simulate ground services and plane handling processes. Ground-based vehicular traffic and the according pathing problems have also been implemented to examine possible solutions to reduce transport costs and emissions while tightening the flight schedule [DZG10]. More specifically, the model was created by gathering empirical data from the aircraft taxiing at the Hamburg Airport, encompassing individual aircraft characteristics and the prevailing environmental conditions. This information was used to establish stochastic relationships that allowed the reproduction and generation of realistic traffic scenarios based on certain parameters. The following functions are featured in the finished model:

- Modular design for easy adjustment of aircraft dynamics, engine performance and airport geometry.
- Engine emission interface for the determination of pollutants during ground operations.
- Determination of taxi process variables such as taxi time and fuel consumption.
- Simulation of conflict situations with the interaction of several aircraft models.
- Extension opportunities for finding optimized taxi routes for given runways and gate positions.

3.3 Model Coupling

The distinct models presented in the previous section have been created by different workgroups, and as such have been implemented with varying modeling paradigms, levels of detail and time resolutions, in different COTS Packages. Furthermore, their development and execution takes place in different geographical locations. It is also important for all the involved parties to be able to access the work done by the various other partners. Hence, the next step in the Airport2030 project was to integrate all these sub-models into a consistent and accessible overall simulation. The following two section will explain why opting for a complex distributed simulation solution like the HLA did not seem wise, but instead a new, simpler method was developed.

3.3.1 Abstraction Level

Wittmann et al. have observed the importance of harnessing the know-how of all the partners and therefore to use the resources already at hand [WHM09]. In the case of this project it was particularly easy, due to the fact the models display a rather abstract level of results in the grand scheme of the cooperation, in spite of their detailed implementations. Typical key figures that are of global interest to the project are as follows:

- Passenger arrivals at the airport
- Passenger and luggage arrivals at the gates
- Turn-around times of the planes

3.3.2 Information Flow

Furthermore, the processes have a distinct forward-oriented flow, with no loops or feedbacks, similar to pipelines. Figure 3.3 shows an overview of how those sub-models are linked in the overall process chain. The transit connection model generates passengers that arrive at the airport terminals at specific times. The passenger and luggage flow in the terminal model simulate the journey of the passengers and their respective baggage items through the terminals to their gates. With the additional data from a flight plan, the ground based processes and flight movements on the apron are then being calculated.

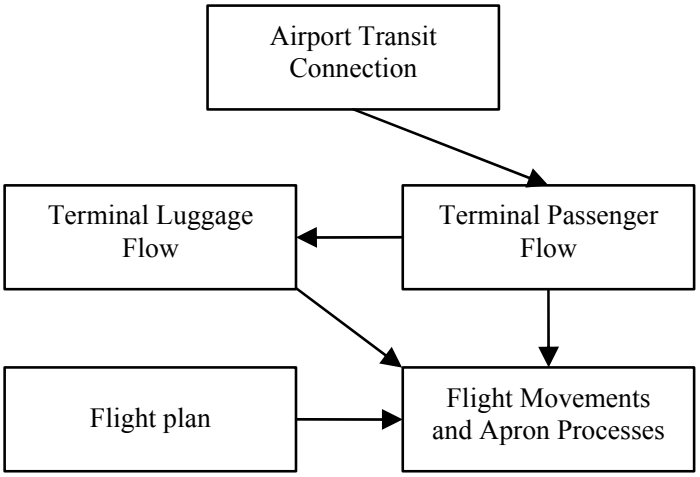


FIGURE 3.3: Information Flow between the Sub-Models of the Airport2030 project

Because of the abstract nature of the metrics used between the sub-models, and their unidirectional information flow, it was renounced to use more complex solutions like the coupling and normalization approaches provided by the High Level Architecture (HLA), and develop the model pipeline approach instead.

3.4 Model Pipeline Prototype

Now that the general parameters of the modeled environment have been clarified, the actual model pipeline prototype is to be described. Following a more technical overview of the implementation of the models, the coupling process as well as the execution of the produced overall simulation is outlined. To complete this section, the inner-workings of the model WSs is analyzed, to gain a better understanding of the processes taking place on the interface level.

3.4.1 Models

As has been mentioned, the various sub-models have been modeled in distinct CSPs:

- *Transit Connection Model:*
  The TUHH is using several complex and resource intensive simulators to generate a detailed representation of the traffic in the metropolitan area of Hamburg. For that reason, it can take up to 24 hours to calculate the passenger flow to the Hamburg Airport with a given set of parameters. Within the scope of this project, it has therefore been decided to reduce the strain on the performance of the execution time of the overall simulation, by choosing a representation of the sub-models results in form of a Structured Query Language (SQL) database [SLA04].
- *Airport Terminal:*
  The UHH decided to model the passenger and luggage flow through the terminals based upon the previously generated arrival rates at the airport in MATLAB SimEvents. SimEvents extends MATLAB with a discrete-event-driven simulation core and a graphical component library. The advantage it offers, is that it easily allows to represent the passenger as entities with attributes.

- *Airport Apron:*

  The DLR is using the flight plan data from a SQL database in conjunction with the outputs from the terminal model to generate the vehicle and plane movements and processes on the apron with the tools of the regular MATLAB environment [MAT11].

3.4.2 Model Pipelines

Model pipelines allow to chain these models together in a hierarchical fashion, as seen in figure 3.4. The leaves of the tree represent WS interfacing with the individual models. These WS have been baptized atomic WS, because of their direct link to a single model of an external CSP. How the interfacing is done on a technical level, is explained in chapter 5. Below them are WS that thematically unite the atomic WS into so called composite WS. As the name implies, these are always a union of at least two interacting WS, exposing their composite service. In this case, the tree first shows a composition into airport terminal WS and airport apron WS, and a on the level below that, the WS executing the entire airport simulation.

The Experiment WS at the root of the tree is a key component of this novel architecture. It serves as a common interface for the operation of simulations coupled through model pipelines. Most importantly, the Experiment WS allows to specify the feedback mechanism to be used for a certain simulation. More on this matter is described in chapter 4. Furthermore, it enables the user to commission several runs of different model pipelines in one execution of the WS. Finally, Experiment WS already offers an interface to execute possible post simulation operations, like the evaluation through external applications.

3.4.3 Simulation Execution

To execute the chain of models composing the airport simulation, an appropriate request has to be sent through the Experiment WS to the Airport WS. Communications with WS is done by sending and receiving SOAP messages. Alongside descriptive information,

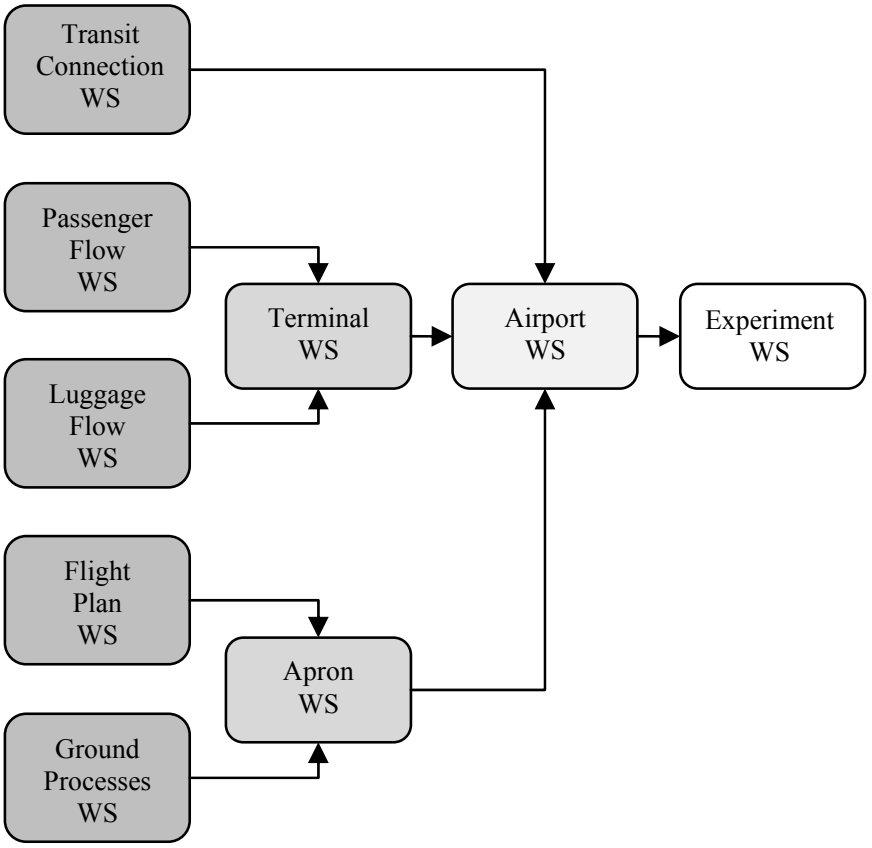


FIGURE 3.4: Model Coupling Hierarchy of the Airport2030 Project

the user must compose this XML-based document according to the predefined input values each atomic WS needs, to instruct the various sub-models through their respective interface. To illustrate this step of the procedure, the basic structure of such a request is shown in the code snippet 3.1. A more complete example of a SOAP request to the WS designed for the Airport2030 project is available in appendix A. It would be tough to show an example of the produced output, as it contains several hundreds of thousands of lines of code. The anticipated structure of the output is therefore included in the provided example.

The Experiment WS has three mandatory parameters: the "address" tag determines the Uniform Resource Locator (URL) of the WS to use during the experiment, "iteration" specifies the feedback handling which will be explained in depth in chapter 4, and "runs" are used to select the models used. Each model needs a certain set of input parameters to produce their output, besides the optional descriptional data.

```
<soapenv:Envelope>
  <soapenv:Header/>
  <soapenv:Body>
    <experiment>
      <date/>
      <description/>
      <address>
        http://127.0.0.1:8080/orbeon/service-airport/
      </address>
      <iteration>
        optimistic
      <iteration/>
      <runs>
        <run>
          <models>
            <model>
              <name>
                airport
              </name>
              <description/>
              <runtime/>
              <parameters>
                <parameter/>
                ...
              </parameters>
              <outputs>
                <output/>
                ...
              </outputs>
            </model>
            ...
          </models>
        </run>
        ...
      </runs>
    </experiment>
  </soapenv:Body>
</soapenv:Envelope>
```

CODE 3.1: Basic Structure of a SOAP Request to the Experiment Web Service

- parameters
  - parameter
    - scenario
    - terminals
      - terminal
        - terminal-number
        - check-ins
          - check-in
            - designation
            - check-in-quantity
            - check-in-capacity
            - check-in-time
            - check-in-luggage
            - check-in-travel-time
    - gates
      - gate
        - gate-number
        - passenger-gate-travel-time
        - luggage-gate-travel-time
    - security
      - automatic-screening
      - manual-screening
      - security-travel-time
      - security-capacity
      - passport-check
      - passport-travel-time
    - luggage
      - automatic-screening
      - level-2-screening
      - luggage-travel-time
      - sorting-time
    - shopping
      - tourist-distribution
      - low-cost-distribution
      - scheduled-distribution
      - shopping-travel-time

TABLE 3.1: Input Parameters for the Airport WS

As the Airport2030 project is still in progress, the specific input parameters of the Airport WS and the output created during the execution are subject to change. A simplified representation of the data used for the airport simulation at the time of writing will however be discussed in the following, to demonstrate the basic makeup of the information handled inside the model pipeline.

Table 3.1 shows that the input parameters of the Airport WS are made up of six main categories. The scenario instructs the transit connection WS and flight plan WS which information to use from their respective databases. A certain scenario specifies the demand for and supply of air travel possibilities. As such it determines the amount of traffic on the transit ways and the arrival rates at the airport, as well as a matching flight plan. The remaining parameters are used in the passenger and luggage WS. The "terminals" tag is used to describe the terminals inside the airport. Their defining characteristic in the abstract model are the check-in stations they provide and how those are operated. The tags "gates", "security" and "luggage" are handled in a similar fashion. "Shopping" caters to a distribution function inside the model, to determine which of the passengers will be spending time shopping.

Table 3.2 shows the output parameters used by the Airport WS. The two main categories here are "passengers" and "flights". Passengers are entities created by the passenger flow model in accordance to the arrival rates at the airport terminals generated by the transit connection model. These display all the information regarding how passengers check in, and the flight they are supposed to catch. Most important here are the different timestamps: "time-terminal" shows the time when a passenger arrives at the terminal, "time-gate" when he has gotten through the terminal and arrives at the gate, and "time-flight" designates when their flight is scheduled to leave. "Flights" holds all the information generated in the apron WS. Again, the time stamps are key to evaluating the processes modeled by the airport apron model.

The process to send the input values to the right models and retrieve the output created is handled by the model pipelines within each WS, starting with the Experiment WS. A representation of the data flow within the system of pipelines can be seen in figure

outputs
output
passengers
passenger
id
date
time-terminal
time-gate
time-flight
stay-duration
check-in-type
luggage-present
airline
gate-cluster
flight-number
flight-type
flight-goal
flights
flight
flight-number
aircraft-type
position
date
passenger-amount
passenger-last
time-scheduled-departure
time-actual-departure
time-delay-departure
time-taxi-out
time-wheels-off
delay-possible

TABLE 3.2: Output Parameters for the Airport WS

3.5. This graphic shows clearly how model pipelines incorporate the pipes and filters architecture paradigm.

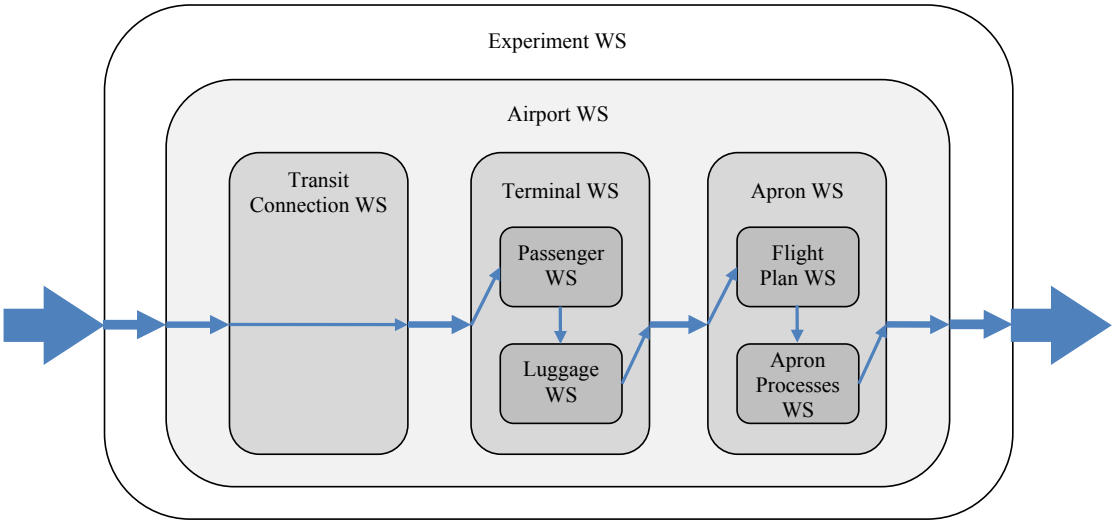


FIGURE 3.5: Data Flow in the Model Pipeline for the Airport2030 Project

In the case presented here, the Experiment WS is instructed to address the Airport WS, and thus forwards all information contained in the SOAP request to it. The Airport WS, as a composite WS, exhibits a chain of commands in its pipeline, which will blindly send to and receive information from its sub- WS in a specific order. This means that, at first, it redirects the request sent originally by the Experiment WS to the first WS in its pipe: the Transit Connection WS. As an atomic WS, its first task is to filter out the input values of interest to the model it interacts with. After that, the WS translates that information into a format readable for the simulator the model was created in. The Transit Connection WS communicates with a SQL database to retrieve all the entries relevant to a distinct scenario. From the input parameters, it therefore only uses the value specified in the "scenario" tag shown in table 3.1. From this value it constructs an SQL command aimed at retrieving the corresponding information from a database. This is done by sending this command as a request to the appropriate database, and retrieving the response it produces, using the standard Transmission Control Protocol (TCP) and Internet Protocol (IP) to communicate. The response is in form of a table containing all the data necessary to continue with the overall simulation. This means everything relating to the "passengers" tag as seen in table 3.2, excluding the arrival times at the gates ("time-gate") and the duration of their stay ("stay-duration"), as this

information has yet to be computed by the passenger and luggage flow models. The Transit Connection WS now transforms this information into XML, adds it as output to the original request, and answers the requesting service client with a SOAP response. In the case shown in figure 3.5, this refers to the Airport WS, which is now able to redirect the response as a request to the Terminal WS. As a composite WS, it behaves analogously to the Airport WS and blindly forwards this request to its sub-WS: the Passenger WS followed by the Luggage WS. As before, these atomic WS each translate the input information they receive, communicate with the simulator of the models they are interfaced with, retrieve an output, package it as SOAP response and send it back to the Terminal WS. The input values used in these two WS is a combination of the original input and the output created during the execution of the Transit Connection WS: both need the information contained in the "terminals", "gates", "security", "luggage" and "shopping" tags (table 3.1), but will also use the data already saved in "passengers" (table 3.2). After the execution of the Passenger and Luggage WS, each "passenger" will now also include data for the tags "time-gate" and "stay-duration". An interesting point here, is that there is no standardized manner for WS, or any application for that matter, to communicate with MATLAB SimEvents, which is the COTS package the passenger and luggage flow models where developed in. To that end, Farschtschi et al. have developed a MATLAB proxy program for the UHH dubbed JaMaLa, which can be communicated with through standard TCP/IP, and can interface with MATLAB and the libraries expanding it [FWH12]. This way, both WS simply need to exchange data with JaMaLa to execute their respective models. More on the possibilities of seamlessly adding applications to interface with different COTS packages will be explained later on. The rest of the pipeline execution follows the same basic procedures which have just been described: the Terminal WS forwards the SOAP request with the newly added output from the Passenger and Luggage WS to the Apron WS, which forwards the request to the Flight Plan WS and then the Apron Processes WS. The Flight Plan WS passes the input value from the "scenario" tag on to its interfaced model, so that it may retrieve a flight plan suitable for that scenario, that then is saved as output in the "flights" section seen in table 3.1, while omitting the "time-actual-departure", "timedelay- departure", "time-taxi-out" and "time-wheels-off" tags for the time being. The Apron Processes

WS then uses exclusively previously generated output values, namely what has been gathered in the "passengers" and "flights" blocks, to fill out the missing data in each "flight" tag. Finally, the Apron WS responds to the request sent by the Airport WS, which then has completed all tasks in its pipeline, and therefore, in turn, forwards a SOAP message to the Experiment WS. The user who executed it with those parameters, is now in essence in possession of the cumulative data provided by all of the five sub-models first presented in figure 3.3.

3.4.4 Web Service Interface

The layered style of architecture model pipelines incorporate becomes apparent in figure 3.6. It demonstrates schematically how the Passenger WS interface with its denominated model in MATLAB SimEvents.

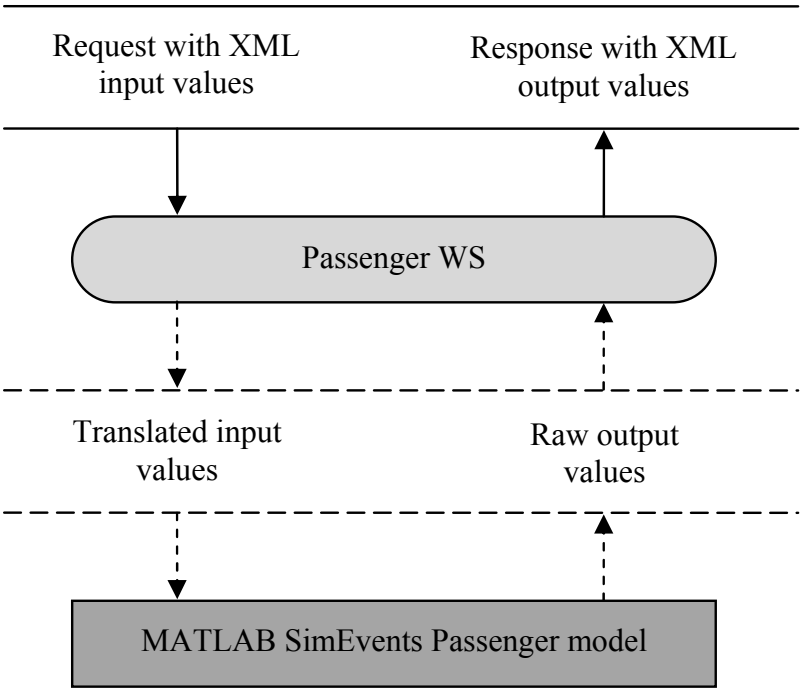


FIGURE 3.6: Passenger WS Interfacing with its External Model

The WS receives a SOAP request from another WS or an external application and processes the information according to its programming. The message is composed of XML data which can easily be sampled for appropriate input values. That information,

if found, is then translated without further interpretation into a format comprehensible for the destined simulator, in this case MATLAB SimEvents. The WS initiates the execution of the model within the simulator according to the translated input. The model thereupon produces a corresponding output, which the WS receives. The data must now be transformed into XML, so that it may be sent to the requesting client as a SOAP response. The procedure is fundamentally the same for each atomic WS.

3.5 Web-Based Interaction

Because of their nature, WSs are a perfectly suited to be controlled by web applications, making it possible to relatively easily offer web-based User Interfaces (UI) to interact with model pipelines. Their advantages shall be illustrated below. Their implementational details as well as their operation is the ulterior subject of this section.

3.5.1 Interface Choice

One further requirement of the Airport2030 project was to develop a usable web-interface so that every involved partner would be able to manipulate the overall simulation. The obvious implementational advantages of web-based solutions to realize distributed simulation have already been outlined: platform independence, maintenance minimization, reusability, interoperability and the ability to share expensive computational resources over the limitations of geographical distance [GSW00]. But the World Wide Web (WWW) has also been the most attractive and important infrastructure in these past years to facilitate the exchange of specialized expert knowledge between collaborating parties. Appelt summarized the advantages that the WWW brings to the table as the basis for tools which support the sharing of information in a collaborative environment [APP99]:

- Browsers provide access to information on the WWW in a platform independent manner.
- Browsers offer a simple and consistent user interface.

- Browsers are an integral part of most operating systems.
- Web servers are commonplace and have become easy to set up and maintain.

Given the circumstances of the Airport2030 project, and in general, situations where the employment of model pipelines becomes valid, it seems web-UIs are the way to go to offer a graphically driven user interaction platform. Web servers and client/server technology allow model pipelines to expose simulation models to the WWW, and web-UIs enables cooperating individuals to experiment with them via an easily accessible common interface.

3.5.2 Characteristics

As it happens, web-UIs are a strong point of the Orbeon Forms platform used in model pipelines, because it implements the W3C XForms standard [XFO09]. XForms is an XML format for the specification of a data processing models for XML and offers users the ability to create interfaces for that data, such as web forms. According to the W3C, the XForms standard exhibits several advantages [XFO03]:

- *Improved usability:*
  It is possible in XForms to specify the data type of the form fields, and to designate which ones are mandatory, or how they relate to each other. The browser is therefore able to check a lot of the inputs for the user, even as the user is making an entry.
- *XML integration:*
  Data collected by XForms is in the XML format and is submitted as XML, enabling true end-to-end XML.
- *Combination of XML technologies:*
  XForms builds upon existing XML technologies, such as XML Path Language (XPath) [XPA99], XML Schema Definition (XSD) [XML00] or Extensible Stylesheet Language Transformations (XSLT) [XSL99], facilitating the implementation of XForms programs.

- *Platform independence:*

  XForms can provide web forms to a wide audience, as they can be delivered as is to nearly all currently established platforms.

3.5.3 Interface Interaction

As WS consume and produce XML data, it is easy to develop appropriate forms for them using XForms. The more so as both, the web forms and the WS, can been authored in the same platform. The result of an effort to create a web-UI in the shape of a web form for the Airport WS can be seen in figure 3.7, with the example being shown of the input fields for the data relevant to the Experiment WS.

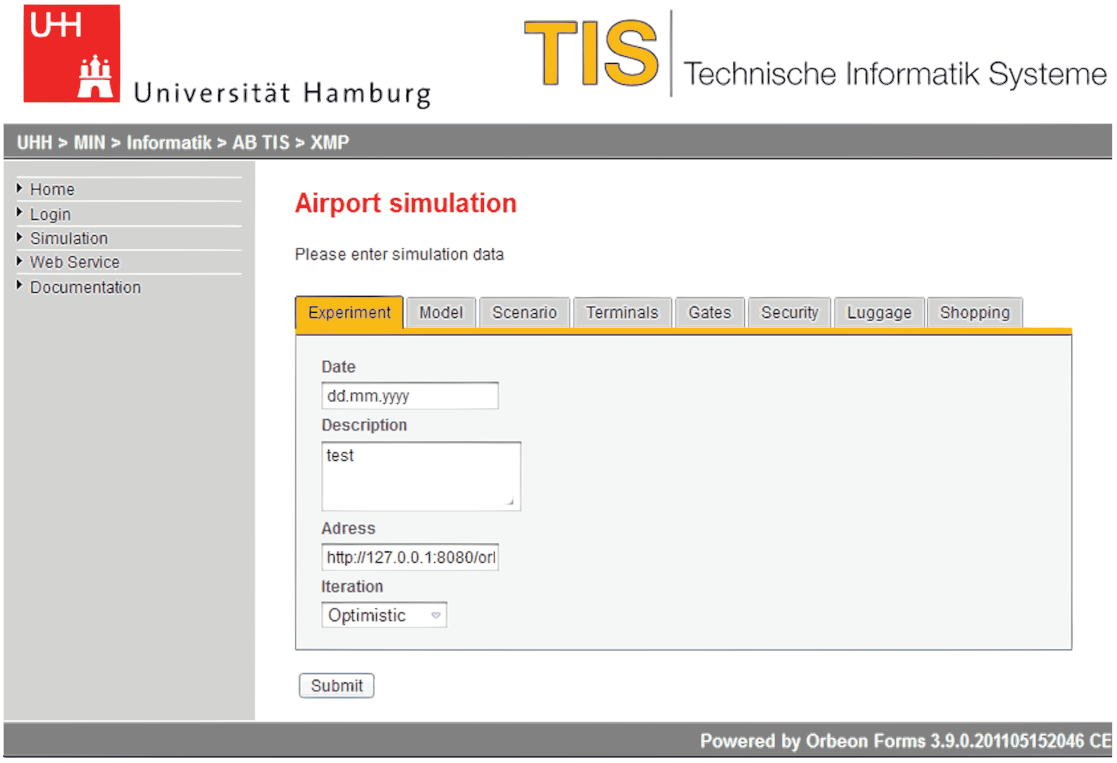


FIGURE 3.7: Input Form for the Airport2030 Project Model Pipeline

Due to the capabilities of XForms, the form is able to build its required input fields in accordance to a standard SOAP request to the Airport WS, much like the one presented schematically beforehand in figure 5. All the major input categories seen there have been implemented as tabs to maximize the clarity of the UI. Each tab displays input fields corresponding to the XML tags shown in figure 6. Complex XML elements like

"terminals" which are composed of several other elements, are displayed using graphical artifacts called accordions, which can be expanded and collapsed at will to further the usability. Since this tag, and the "check-ins" it includes, can appear multiple time per request, the functionality to add or remove instance of those inputs has been implemented. An idea of how this is used in the Web-UI can be seen in figure 3.8.

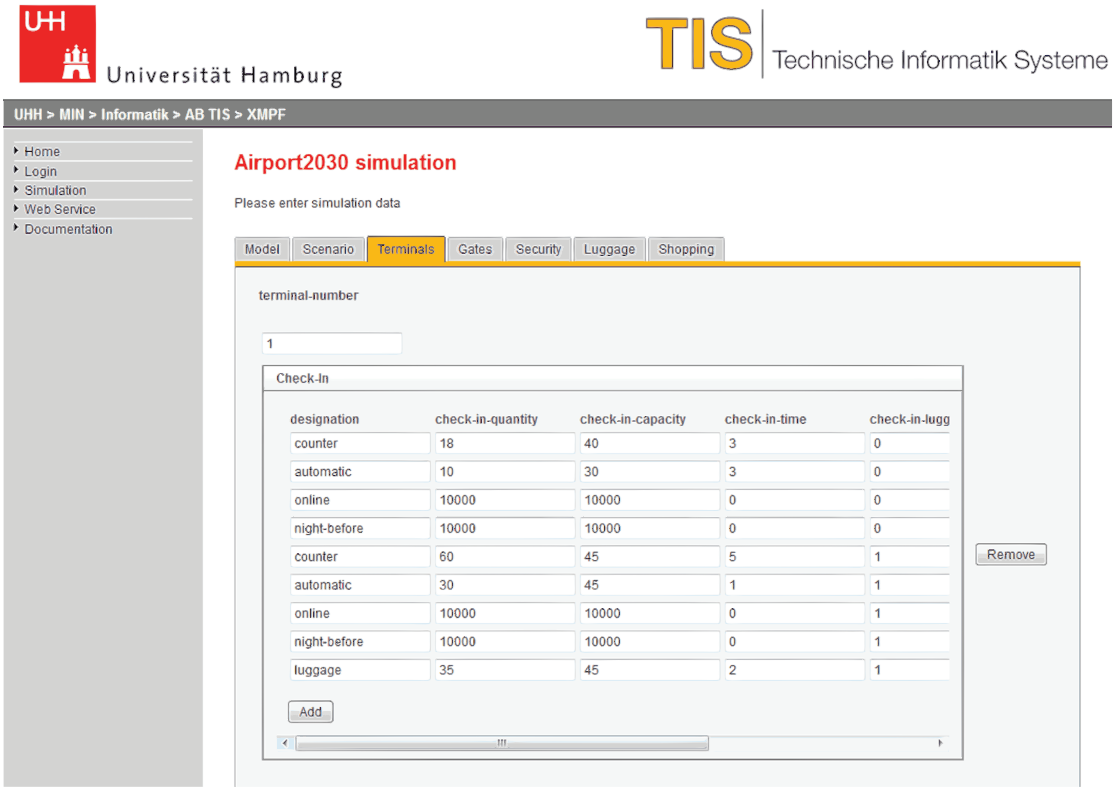
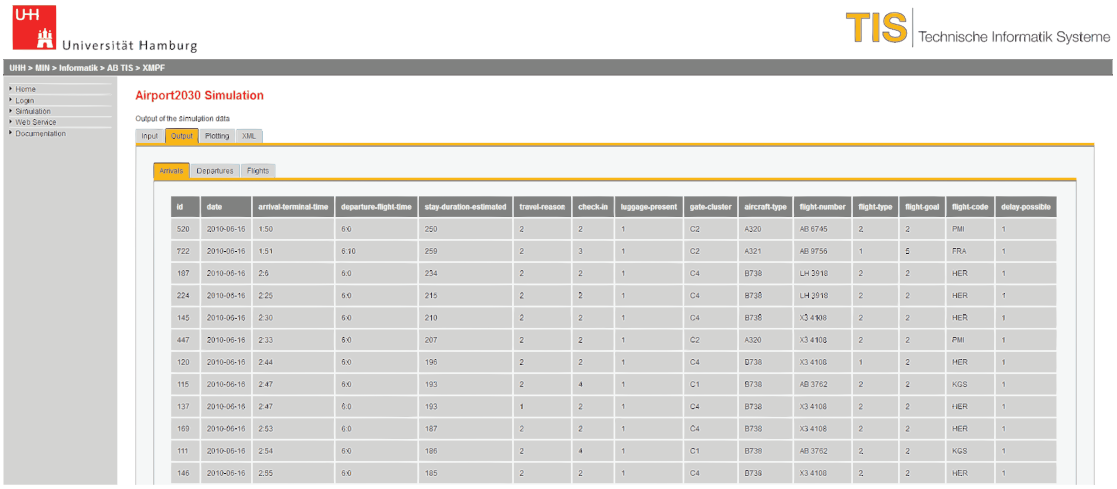


FIGURE 3.8: Complex Element Display in the Input Form

Once all inputs have been entered in the form, the user can issue his request to the Airport pipeline by activating the "submit" button. The XForms page now converts the values supplied to the form into an actual SOAP request, which is sent to the Experiment WS (and consequently forwarded to the Airport WS), as it would happen if the WS would have been addressed by any other third party application. Once the distributed simulation has been executed, the Experiment WS responds to the original requestor with a response containing the results, as expected. The Web-UI reacts to this response by changing the page view from an input form to a categorized view of the results as seen in figure 3.9.



id	date	arrival-terminal-time	departure-flight-time	stay-duration-estimated	travel-reason	check-in	luggage-present	gate-cluster	aircraft-type	flight-number	flight-type	flight-goal	flight-code	delay-possible
520	2010-06-16	1:50	6:0	250	2	2	1	C2	A320	AB 6745	2	2	PMI	1
722	2010-06-16	1:51	6:10	259	2	3	1	C2	A321	AB 9756	1	5	FRA	1
187	2010-06-16	2:6	6:0	234	2	2	1	C4	B738	LH 3918	2	2	HER	1
224	2010-06-16	2:25	6:0	215	2	2	1	C4	B738	LH 3918	2	2	HER	1
145	2010-06-16	2:30	6:0	210	2	2	1	C4	B738	X3 4108	2	2	HER	1
447	2010-06-16	2:33	6:0	207	2	2	1	C2	A320	X3 4108	2	2	PMI	1
120	2010-06-16	2:44	6:0	196	2	2	1	C4	B738	X3 4108	1	2	HER	1
115	2010-06-16	2:47	6:0	193	2	4	1	C1	B738	AB 3762	2	2	KGS	1
137	2010-06-16	2:47	6:0	193	1	2	1	C4	B738	X3 4108	2	2	HER	1
169	2010-06-16	2:53	6:0	187	2	2	1	C4	B738	X3 4108	2	2	HER	1
111	2010-06-16	2:54	6:0	186	2	4	1	C1	B738	AB 3762	2	2	KGS	1
146	2010-06-16	2:55	6:0	185	2	2	1	C4	B738	X3 4108	2	2	HER	1

FIGURE 3.9: Output View for the Airport2030 Project Model Pipeline

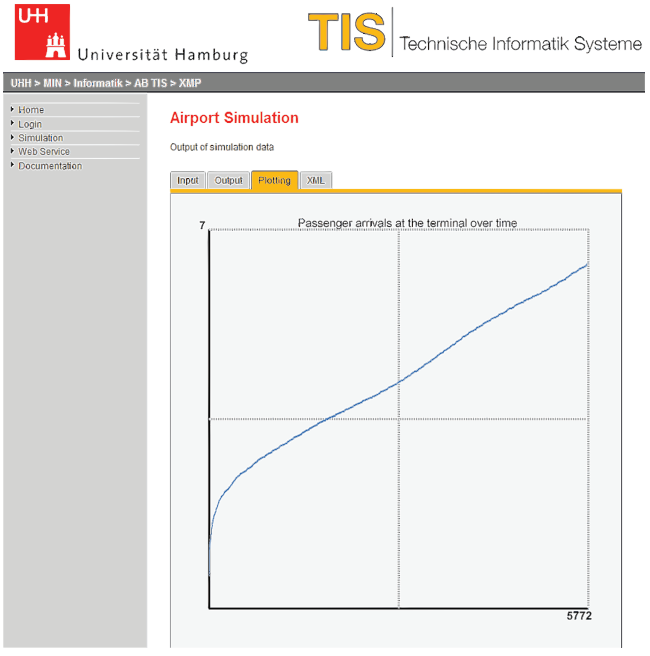


FIGURE 3.10: SVG Graph Display in the Output View

The interface is able to display the information in that fashion, by relying heavily on applying XSLT operations on the received SOAP response. After execution of a WS, the web-UI shows the input and output values, as well as plotted graphs and the raw XML data in different tabs. The input and output tabs contain a tabular view of the information that was entered by the user and produced in the simulation respectively. The "plotting" tab displays a selection of values as graphs. This is made possible by transforming portions of the data contained in the SOAP response into the Scalable Vector Graphics (SVG) format, which is a language for describing two-dimensional vector graphics in XML [SVG11]. Since the data from the model pipeline is already in XML, the conversion poses no problems, as shown in figure 3.10.

Lastly, the output view the web-UI offers, can also display the raw XML information that was exchanged during the execution of the model pipeline, and allows to save that data locally as an XML file, for use in third party applications (figure 3.11).

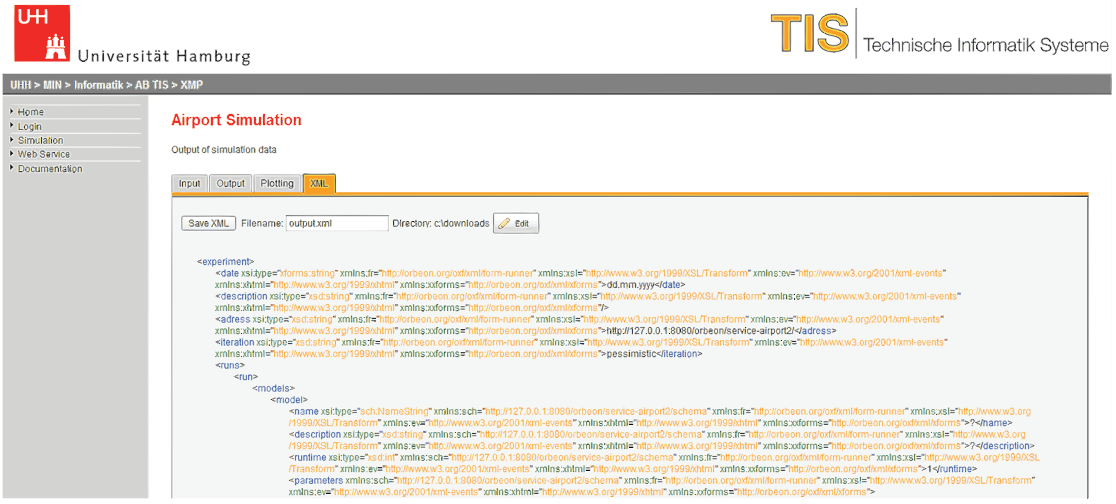


FIGURE 3.11: Raw XML Display in the Output View

Chapter 4

Feedback Capabilities of Model Pipelines

4.1 Basic Concepts

This chapter deals with the feedback capabilities of the model pipeline approach. To properly introduce the subject, the first order of business is to summarize the most basic concepts relevant to the subject in relation to the simulation field. The first step is to distinguish between the three notions of time that are commonly used in the context of simulation. According to Fujimoto, these are defined as follows [FUJ00]:

1. *Physical time:*
   The time in the physical system being modeled (e.g. 01.01.2013).
2. *Simulation time:*
   The abstraction of time used in the simulation to represent the physical time (e.g. the number of minutes in a specific day, modeled as integer value).
3. *Wall-clock time:*
   The real time that has elapsed during the execution of a simulation (e.g. the number of elapsed milliseconds since the start of the simulation).

Obviously, the relation between physical time and wall-clock time is entirely dependent upon the simulation, the execution parameters and the hardware environment. Physical time is generally mapped one-to-one to simulation time. But simulation time may not exhibit any specific relationship with wall-clock time. This depends on the mode of execution of the simulation:

1. *Scaled real-time execution:*

   Scaled real-time executions have a simulation time that is linear proportional to the wall-clock time. This means that the simulation time is slower or faster than the wall-clock time by a constant factor.
2. *Real-time execution:*

   This mode of execution is a special case of the scaled real-time execution, because he simulation time advances in sync with the wall-clock time. For each time unit of time elapsed in simulation time, the wall-clock time also advances one unit.
3. *As-fast-as-possible execution:*

   As the name suggests, in as-fast-as-possible executions the simulation is executed as fast as the computing speed of the hardware allows it to. Therefore, no relationship between simulation and wall-clock time is maintained.

The manner in which states are changed within a model while advancing simulation time, is a further classification subject of importance. In discrete simulation, there are two so called time flow mechanisms that are most commonly used [FUJ00]:

1. *Time-stepped execution:*

   In this case, the simulation time is sub-divided in equal-sized time steps, which the simulation uses to advance its execution. Each time step, a new state for the simulation is calculated, even if it remains unchanged.

2. *Event-driven execution:*

   To avoid computing a new state in each time step, it is generally more efficient to wait for the execution a specific event in the simulation. An event is an abstraction of a real-life process occurring in the physical system that the simulation is supposed to represent. The point in time when the event occurs is marked by a timestamp within the simulation.

In distributed simulations, the exchange of events between processors has to be carefully coordinated. To this end, there are multiple types of delivery mechanisms that can be applied at runtime, with two of them being implemented more commonly [PER06]:

1. *Receive-order delivery:*

   The events sent by various processors to a receiving processor are ordered by the time they have been received. This method has the advantage that the exchange of events generally has a lower latency in comparison to timestamp-order systems.

2. *Timestamp-order delivery:*

   This approach requires events received by a processor to be buffered first, to undergo a runtime check to order the received events in a specific time frame, according to their timestamp in an ascending fashion. While this comes with the price of higher latencies, the timestamp-order delivery system can guarantee the temporal ordering of events, and thus preserve their potential causal relationships.

4.2 Synchronization Techniques

With the basics established, this thesis will now touch upon the important concept of synchronization. In distributed (and parallel) simulation theory, the use of spatial decomposition is the most widely applied technique (in contrast to temporal decomposition) [PER06]. In this approach, a system is viewed as a composition of interacting physical processes. These are modeled as logical processes and the interactions between them are made possible by the exchange of time-stamped events. Each logical process

performs a sequence of computations referred to as events, which may lead to an alteration of state variables or the scheduling of new events. What has come to be known as synchronization problem, describes the issue that every logical process is required to execute its events in time stamp order, even those generated by other processes. In response to this problem, several approaches have been developed, of which the most prevalent are presented in this section.

4.2.1 Conservative Approach

Conservative synchronization algorithms strictly avoid processing events out of time stamp order [CHM79]. To that end, logical processes have to guarantee that no new event can be received with a smaller time stamp as the last computed one. Either such processes execute only the event with the globally smallest timestamp, or potential future interactions between processes must be predicted. The way to accomplish this, is to determine a property called lookahead for a simulation, which designates a time window specific to every simulation that uses a conservative synchronization mechanism, where it is safe to compute existing events, because no new events can be generated by other processes with a smaller time stamp. This property is however rather hard to establish with increasingly complex simulations [PRL89], although the implementation of its associated algorithm is comparatively easy in regards to other synchronization techniques.

4.2.2 Optimistic Approach

Optimistic synchronization algorithms do not block processes to make them wait for the expiration of a lookahead window [JEF85]. Instead, the simulation assumes that the time stamps are in order and no causality errors can occur. Events can therefore be processed when they are first created, allowing the lookahead to even be defined as zero, without affecting the runtime performance too much [PER06]. In the event of a temporal inconsistency, optimistic synchronization mechanisms provide compensation procedures such as rolling the simulation back to a safe state, to undo modifications

to state variables. This method, called state-saving, logs the state of these variables at specific checkpoints, before changes can affect them. When an error has been caught and computations have to be negated, the simulation can fetch the previous safe state variable from that checkpoint log. Alternatively, with reverse computation, it is also possible to roll back the simulation to a safe state by invoking perfect inverses of the event computations that changed the state variables [CPF99]. The issue of extracting a viable lookahead value is eliminated with this synchronization approach, but instead compensating code has to be implemented for the described roll back procedures.

4.2.3 Relaxed Approach

With relaxed synchronization, the processing of events is not strictly bound to their temporal order [RTR98]. If two or more events exhibit almost identical time stamps, it becomes acceptable to compute either of them. Although this approach provides a simpler implementation of a synchronization algorithm, the validity and repeatability of the simulation can become an issue. Because in this case, it may be possible for multiple executions of the simulation to output differing results, special care has to be taken for every simulation individually to foreclose such effects.

4.2.4 Combined Approach

There is also an approach called combined synchronization, which combines elements of the previous algorithms. Parts of a simulation can be execute conservatively because the compensation mechanisms are hard to implement or the lookahead has a large value. While another part of the simulation can be executed optimistically when an appropriate lookahead is hard to assess, or so low that it may affect computation time too greatly [RAT93]. The HLA actually is a prime example of such a combined approach to synchronization.

In the case of heterogeneous distributed simulations, it is often the case that the various involved models feature varying levels granularity and different model paradigms.

To force the use of a single synchronization method onto one such simulation, would imply either that the modelers have to convene to determine a specific approach before coupling their models, or the reimplementation of a number of the models to match the new synchronization algorithm. Furthermore, any new additions to the underlying coupling framework would cause costly overhauls to the whole system. These are clearly unfavorable conditions for distributed simulation in heterogeneous environments, like in joint projects. Therefore, frameworks that incorporate several synchronization approaches have been developed, that promise to remain impervious to changes to the distributed simulations for which they actuate the coupling. The HLA for example, is such a framework. It has been designed to accommodate multiple simulator types, which however has made the HLA more suited to couple coarse-grained simulators, and makes the high-performance execution of fine-grained simulations more challenging according to Perumalla [PER06].

4.3 Coupling Mechanisms

Now that a better understanding of the most widespread synchronization concepts have been established, this section will describe what the classical and model pipeline coupling mechanisms look like, as well as what their differences are. This will illustrate the advantages and drawbacks of the novel approach presented in this thesis.

4.3.1 Classical Coupling

Shared variables and message passing are the two dominant forms of communications used in parallel and distributed simulation [FUJ00]. Shared variables can obviously be accessed through shared memory, while message passing algorithms can be implemented using shared memory by implementing shared queues to hold the exchanged messages. Therefore, on shared memory machines, distributed event driven simulations are analogous to the execution of parallel programs, because events that propagate

between processes of a simulation, can be seen as accesses to variables of a shared memory [RIC95].

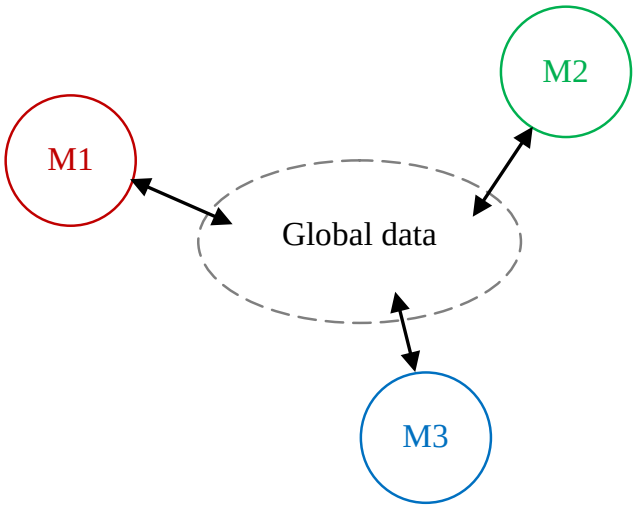


FIGURE 4.1: Data Access in Classical Coupling Mechanisms (confer [WHM09])

Figure 4.1 depicts the usual way in which multiple models exchange information in a distributed environment. That data can be described as "global", because it is freely accessible to all of the models, as if they were accessing it on a shared memory system. That global data area permits all models to communicate with one another with no restrictions but those imposed by the employed synchronization approach.

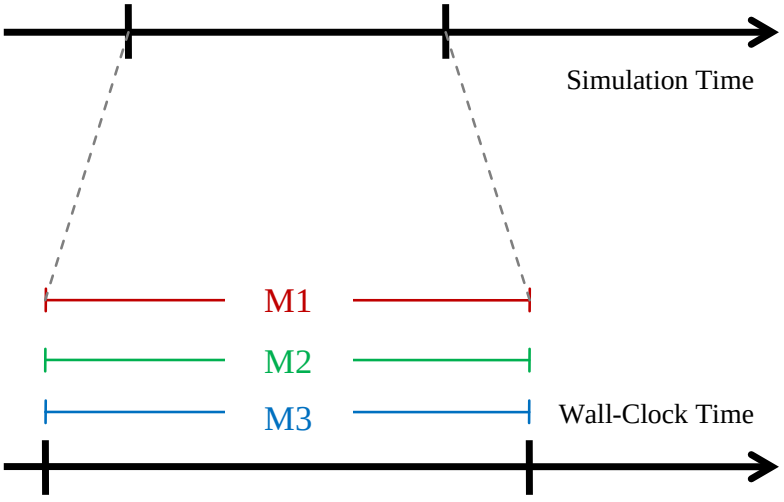


FIGURE 4.2: Overlapping Wall-Clock Times in Classical Coupling Mechanisms (confer [WHM09])

As has been discussed previously, the runtime system has generally the task to synchronize the progression of the local simulation times of each model in correlation with the wall-clock time, to warrant the sanity of the local and global data being handled. In classical coupling methodology, simulating a specific time frame with a time-stepped execution implies that all involved models will run concurrently, meaning that the simulation times of the different models will overlap, as seen in figure 4.2.

4.3.2 Pipeline Coupling

Model pipelines however, have a unique way to avoid having to employ advanced synchronization algorithms.

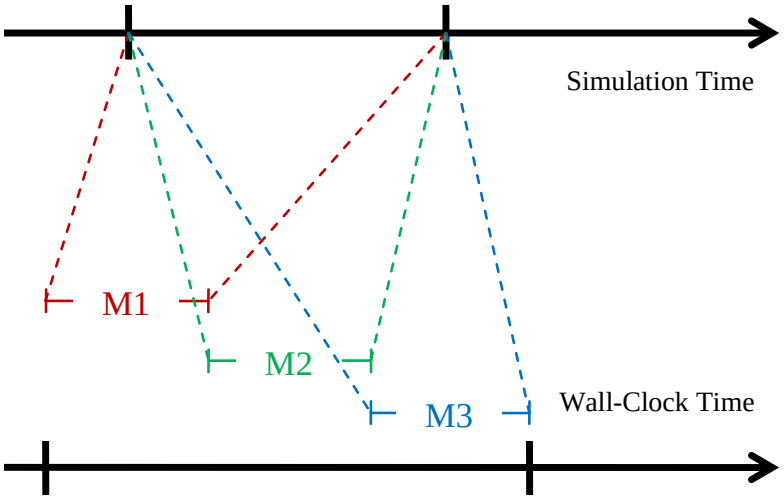


FIGURE 4.3: Distinct Wall-Clock Times in the Pipeline Coupling Mechanism (confer [WHM09])

Figure 4.3 demonstrates that contrary to classical coupling, the wall-clock times of the individual models in a model pipeline do not overlap during the same simulation time, because they are executed sequentially. The input fed to the distributed simulation is constantly being expanded upon by adding and substituting information from the execution of the individual models until an output has been produced, as seen in figure 4.4. It makes therefore no sense to use synchronization points to ensure that state variables have been altered correctly.

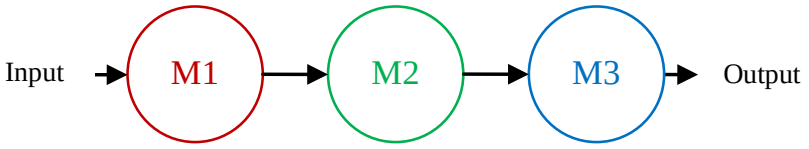


FIGURE 4.4: Data Access in the Pipeline Coupling Mechanism (confer [WHM09])

M1 has to be executed in its entirety and compute its output for the modeled simulation time for the overall simulation to advance. Only then is it possible for M2 to be executed. Similarly, M2 has to finish its execution before M3 can accept an input at its interface. Himstedt and Wittmann have therefore identified that models in a pipeline have a so called "before" relation [HIW10].

Obviously, this does not mean that models of a model pipeline do generally not need to be synchronized in some manner. The problem is that the message passing capabilities of model pipelines is somewhat hampered by the sequential nature of their architecture, and therefore the global sanity of state variables is not always guaranteed. It is not possible for events to be processed out of time stamp order by a model, as long as they have been generated in a model previously executed in the pipeline, since it was executed to completion, and therefore all necessary information to ensure proper processing is available to all following models from the get go. The situation is more intricate if a model is generating events of interest to other models situated before itself in the pipeline. This becomes obvious when observing figure 4.4: all events generated in M1 and M2 will be known to M3 when it is executed, and no inconsistencies with its state variables should occur. But if M3 now generates new events relevant to M1 and M2, the output they previously produced would very likely be faulty. Since this is unacceptable for a distributed simulation architecture, two feedback algorithms have been implemented to allow models to seamlessly communicate within pipelines, and thus avoid conflicting data.

4.4 Feedback Algorithms

The feedback algorithms in model pipelines are loosely based on the principle of rolling back simulations to restore them to a consistent state. It has been explained that optimistic synchronization solutions usually employ such an approach in conjunction with state-saving methods. These provide mechanisms to keep a copy of the overall state of the system at certain intervals or before the computations of particular events. If the system has to be rolled back due to data inconsistencies, one of the restore points created beforehand can be used. According to Perumalla however, the management of these points can cause a non-negligible memory overhead [PER06]. Already their creation turns out to be extremely complex. It is thusly not surprising that currently even a lot of CSPs do not provide sophisticated mechanisms to memorize a model's state or provide the ability to return to previous states [SSL07].

4.4.1 Iteratively Driven Simulation

The idea of Himstedt and Wittmann is to use the start of the simulation as the only restore point of a model pipeline simulation [HIW10] and advance its simulation time run by run, thereby avoiding the management of universal restore points.

Simulation Time →

	0…1	1…2	2…3	3…4	4…5	*n*-1…*n*
Run 1	Input → M1 → M2 → M3 → Output[0…1]					
Run 2	Input ∪ Output[0…1] → M1 → M2 → M3 → Output[0…2]					
Run 3	Input ∪ Output[0…2] → M1 → M2 → M3 → Output[0…3]					
Run 4	Input ∪ Output[0…3] → M1 → M2 → M3 → Output[0…4]					
Run 5	Input ∪ Output[0…4] → M1 → M2 → M3 → Output[0…5]					
Run *n*	Input ∪ Output[0…*n*-1] → M1 → M2 → M3 → Output[0…*n*]					

TABLE 4.1: Iteratively Driven Simulation (confer [WFH12])

Table 4.1 serves to illustrate this principle, called Iteratively Driven Simulation (IDS), for a distributed simulation involving three distinct models M1, M2 and M3. In the first run, the simulation is executed for the interval between the time units 0 and 1 of

the overall simulation time, with the initial parameters as input. The output created that way is identified as Output[0...1]. In the second run, those output values are added to the initial parameters to form the new input. This time, the interval between the time units 0 and 2 is simulated. This allows to repeat the previous run, while all the models have the results generated between the time units 0 and 1 already at their disposal. Through that feedback mechanism, models gain the ability to exchange information with preceding models of the pipeline. In this specific example, the second iteration of the simulation enables M3 to relay any relevant event generation logged in Output[0...1] created in the first iteration to M1 and M2. The principle is the same for all following runs. Generally speaking, the simulated simulation time frame is incremented by one time unit in each run. That allows every model in the run n to have access to the results of Output[0...*n*-1], even if those results contain data generated by models executed after them in the layout of the pipeline. To avoid data inconsistencies, it is however not possible for any model to use information that was generated in the future from its perspective: in the fifth run for example, M1 cannot access the output generated by M3 in the time interval between 3 and 4, when itself is still computing the results of the time interval between 2 and 3.

Because of the reiteration of runs during the simulation, the wall-clock time increases with the amount of necessary simulation runs, and is therefore directly correlated to the simulation time. Thus, the advantage of forgoing state-saves and restore points comes at the cost of execution time, as well as additional data transport costs. This represents an unnecessary hindrance for simulations with few or no need for feedbacks. For this reason Widemann et al. have developed the Optimistic Iteratively Driven Simulation (OIDS) [WFH11].

As has been discussed previously, to execute a simulation with model pipelines it is necessary to send a SOAP message to the Experiment WS with the appropriate information. Alongside the input parameters, this specific WS also requires instructions on how to perform feedbacks, as seen in the code snippet 3.1. The "iteration" tag allows for two possible input strings: "pessimistic" and "optimistic". While entering the word "pessimistic" signalizes the Experiment WS to use the IDS approach, "optimistic" serves to execute the OIDS principle.

4.4.2 Optimistic Iteratively Driven Simulation

The OIDS is pursuing the idea of equipping the experiment WS as an administrative component with an additional functionality, so it can decide for itself when a run has to be repeated. For this purpose, the WS analyzes the output of the overall simulation and initiates a targeted search for feedback events without ever interpreting the values of the XML content, which respects the black-box nature of the sub-models. The description of relevant events feedback is handled by the simulation developers using appropriate conventions at the interfaces of the sub-models.

```
<feedbacks>
  <feedback>
    <step/>
    <ws/>
    <tags>
      <tag>
        <name/>
        <value/>
      </tag>
      ...
    </tags>
    <processed/>
  </feedback>
  ...
</feedbacks>
```

CODE 4.1: Complex XML Data Type for Feedback Events

The code snippet 4.1 shows an example of how feedback events are described and signaled in model pipelines using XML tags. The associated XML data type contains the information to process feedbacks within a pipeline model. It includes the simulation step in which the feedback is generated ("step"), which WS created the feedback event ("ws"), the tags from the affected target WS ("name"), the corresponding value ("value"), and an indication of whether the feedback event has already been processed or not ("processed").

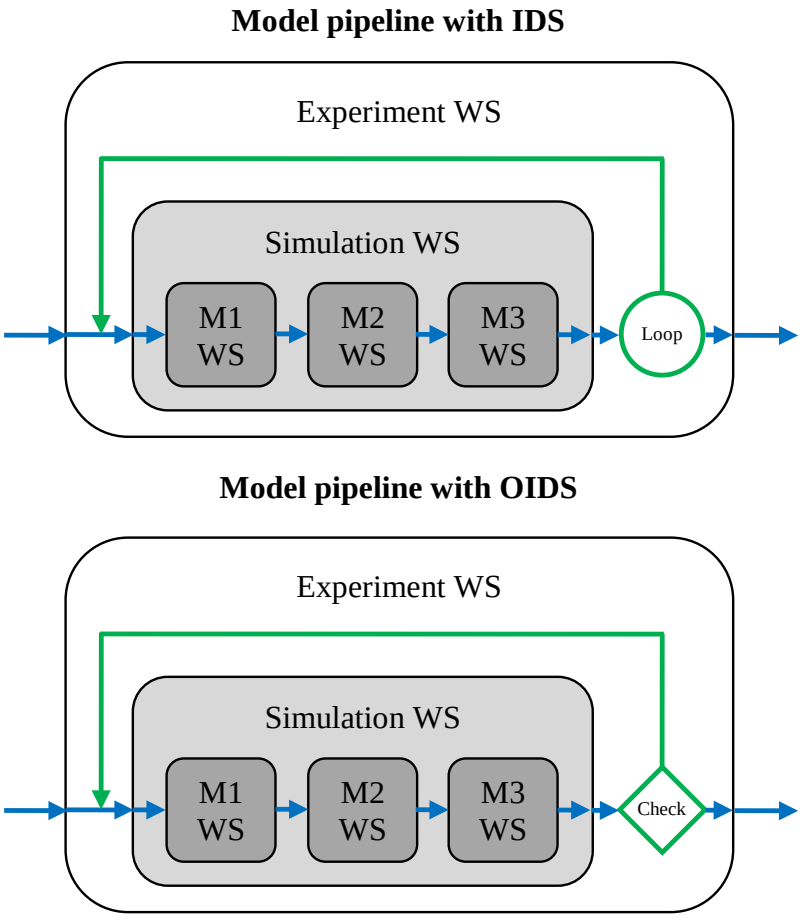


FIGURE 4.5: Comparison of the IDS and OIDS Approaches

Figure 4.5 illustrates the data flow inside of a model pipeline simulation, first with the feedbacks being handled by the IDS approach, then with OIDS. The algorithm for IDS works as follows:

1. The input data is specified over the Experiment WS.
2. The Experiment WS passes the information to the Simulation WS, to initialize the actual simulation.
3. The Simulation WS passes the information to its sub-WS, so that the data may get processed sequentially.
4. The data is returned by the Simulation WS to the Experiment WS, which then monitors if the necessary numbers of iterations have been executed.

5. If the simulation still needs reruns, the initial input data is expanded with the current output and is fed back into the Simulation WS. The algorithm continues with step 2.

6. Otherwise, the Experiment WS delivers the final results of the simulation.

With the OIDS approach, the last steps of the process differ slightly. There is no fixed number of iterations that the Simulation WS has to go through. Instead, the Experiment WS checks if one of the Sub-WS has generated any feedback requests, by analyzing the produced output:

4. The data is returned by the Simulation WS to the Experiment WS, which then checks if the output contains feedback requests, without evaluating the values however.

5. If the output contains unprocessed feedback events, the initial input data is expanded with the current output and is fed back into the Simulation WS. The algorithm continues with step 2.

6. Otherwise, the Experiment WS delivers the final results of the simulation.

Faster execution times compared to IDS are to be expected in particular for simulations with a small number of feedback events. But there are still situations where the IDS principle would perform better than OIDS. The following section will identify those situations through a series of measurements.

4.5 Measurements Comparison

The test environment used for the measurements in this section is the prototypical model pipeline for the simulation of airport-related processes developed for the Airport2030 project discussed in chapter 3. The airport simulation is designed to simulate the processes in and around an airport for one whole day for varying parameters. A relatively

coarse granularity has been chosen for this simulation by partitioning the day into hourly time steps, i.e. a total of 24 time steps have to be computed to fully execute the simulation. All values were determined using soapUI, a comprehensive and widely used tool for testing WS [EVI11]. As a reference, the specifics of the computer with the test environment are listed below:

- Windows 7 Professional 64-bit
- Intel Core i5 750 @ 2.67 GHz
- 8 GB DDR3 RAM

4.5.1 Measurements

At first, it is important to examine the additional time needed to execute a full run of the simulation with IDS, in comparison to an execution without feedback capabilities. In order to illustrate how the wall-clock time increases with the number of iterations, the model pipeline has been executed several times, each time with decreasing time step sizes, thus increasing the number of reruns. Since 1, 2, 3, 4, 6, 8, 12 and 24 are all multiples of 24, they are valid step sizes for the Airport2030 simulation. The quotients of the various step sizes with 24 represent the amount of iterations necessary to complete the simulation in its entirety. A summary of the results produced by this series of measurements is shown in table 4.2.

An execution with a step size of 24 and only one iteration produces a wall clock time as low as 9,2 seconds. But an execution with three runs by choosing a step size of 8 already doubles the wall-clock time, increasing it to 19,8 seconds. This trend can observed up to the execution of the simulation with a step size of 1, thus forcing it to run 24 times, which puts the wall-clock time at 124.9 seconds. The graph in figure 4.6 clearly demonstrates how the computation time linearly increases in direct proportion to the amount of iterations necessary to execute the simulation. For simulations with a long simulation time and high granularity, translating to small time intervals to compute for a numerous amount of runs, the wall-clock time can become quite big relatively fast.

Number of iterations	Wall-clock time (in seconds)
1	9,2
2	14,3
3	19,8
4	25,2
6	35,3
8	44,8
12	64,7
24	124,9

TABLE 4.2: Wall-Clock Times of IDS with Growing Number of Iterations

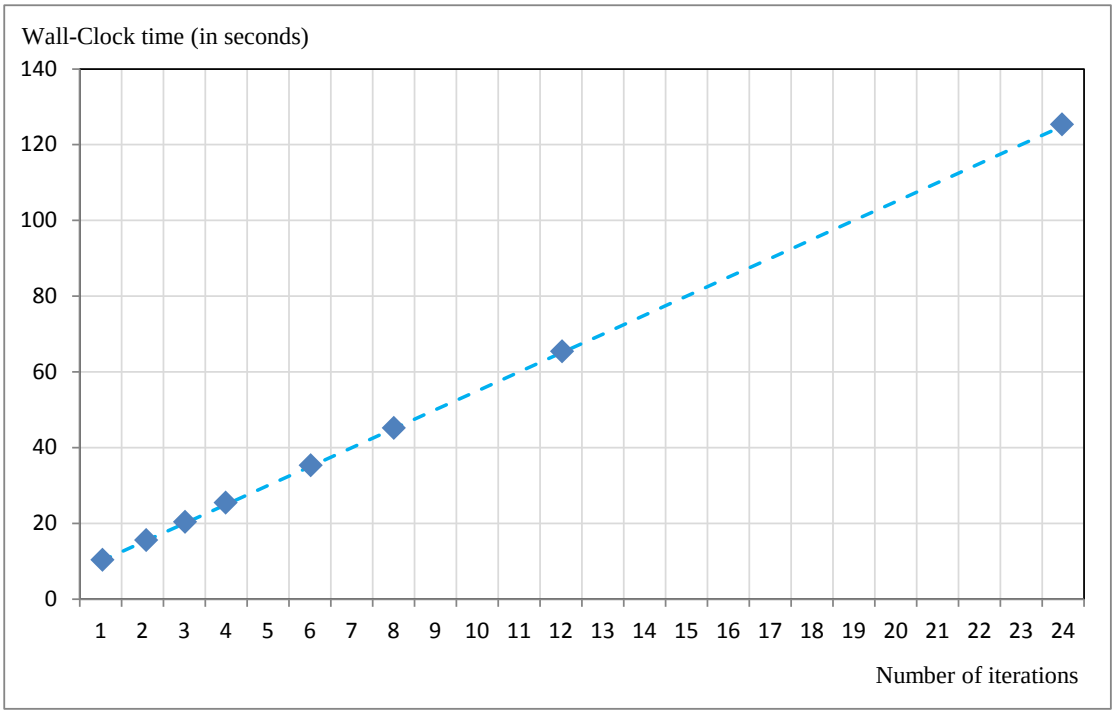


FIGURE 4.6: Growing Wall-Clock Times with Increasing Number of Iterations for IDS

The OIDS approach on the other hand, will only initiate reruns of the simulation each time a feedback event was generated. In the next series of tests, the simulation was therefore executed with artificially induced feedbacks to successfully regulate the measurements. On the first execution with OIDS no feedbacks requests where created, after that however the simulation contained 6, 12, 18 and 24 requests respectively for each measurement series. For the sake of simplicity it is assumed within this study that only one feedback event can occur during each run of the simulation, i.e. that the number of feedbacks corresponds to the number of iteration cycles in one execution of the Experiment WS. The comparative values used for the IDS approach come from the already presented measurements for it, with the minimal simulation time step size of 1. One can see in table 4.3 that for simulations with OIDS and without or few feedbacks, the wall-clock times are significantly lower than with IDS, while the IDS approach works faster if 12 or more feedback events occur in a simulation with OIDS.

Feedback algorithm	Wall-clock time (in seconds)
IDS	124,9
OIDS with 0 feedbacks	10,4
OIDS with 6 feedbacks	73,7
OIDS with 12 feedbacks	138,3
OIDS with 18 feedbacks	179,1
OIDS with 24 feedbacks	247,2

TABLE 4.3: Wall-Clock Times of IDS and OIDS

To clarify, the average wall-clock times are shown again in the form of a column chart in figure 4.7. The first column corresponds to the wall-clock time of the model pipeline with IDS, and the remaining columns show the runtime with OIDS with an increasing number of feedback events.

4.5.2 Analysis

Since the model pipeline using IDS has to execute exactly 24 runs regardless of the actual number of generated feedback requests, the wall-clock time remains constant.

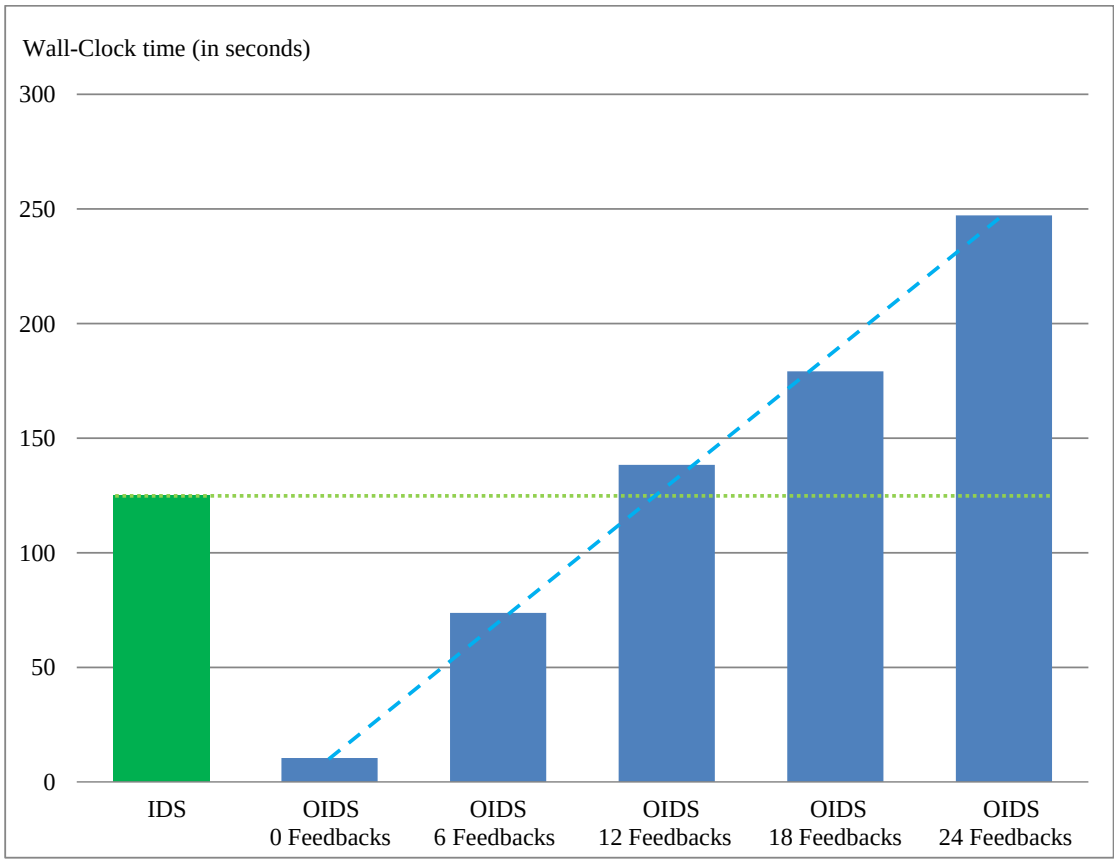


FIGURE 4.7: Comparison of the Wall-Clock Times of IDS and OIDS

However, a single run of the model pipeline will not simulate the whole simulation time of 24 hours, but instead increment the simulated time frame step by step, as seen in table 4.1. By leaving aside the comparatively negligible overhead produced by the increased communication needs, the wall-clock time L can be calculated by using the Gaussian sum formula:

$$L = \left(\frac{a}{n}\right) * 1 + \ldots + \left(\frac{a}{n}\right) * n = \sum_{k=1}^{n} \left(\frac{a}{n}\right) * k = \frac{1}{2} * a * (n+1) \tag{4.1}$$

Where a is the duration of a single simulation run with the maximum simulation time, and n is the maximum number of runs at the selected step size. The wall-clock times of the model pipeline with OIDS are dependent on the number of feedback events, which is why they increase steadily with the growing numbers of triggered iterations. If the

additional communications overhead is ignored again, the wall-clock time L can be calculated via the following formula:

$$L = a + a * x = a * (1 + x) \tag{4.2}$$

Where x is the number of runs necessary to process all feedback events. To now determine when the OIDS approach is superior to the IDS, the solution to the following equation has to be found:

$$\frac{1}{2} * a * (n+1) = a * (1+x) \Leftrightarrow x = \frac{n-1}{2} \tag{4.3}$$

This result is confirmed by the observations of figure 4.7. For the prototypical implementation of the model pipeline $l = 24$ In that case $x = \left(\frac{23}{2}\right) = 11.5$. The fourth series of measurements indeed shows that the execution time for OIDS with 12 feedback requests actually only is slightly larger than for the first series of measurements with IDS. The application of a model pipeline using OIDS is therefore useful until the number of runs needed to process all the feedbacks exceeds half the maximum number of iterations IDS would produce for the smallest simulation time step size.

Chapter 5

The Model Pipeline Framework

5.1 Runtime Environment

In the two previous chapters, this thesis has explained how model pipelines operate. This chapter will now present an in-depth analysis of the model pipeline framework used to build them, mostly studying the conceptual and implementational details of this new technology. To introduce the subject, this section will describe the runtime environment model the framework is based upon.

As has already been pointed out, the Orbeon Forms platform has been chosen as the basic frame to build the XPL Model Pipeline Framework (XMPF) [ORB11]. Orbeon Forms is an open source, standard-based web forms solution, which is built around a user-friendly Ajax-based XForms engine. It also implements a mature, high-performance XML pipeline engine for the processing of XML data. Therefore, this architecture emerges as perfectly suited option for the tasks of capturing, processing and presenting simulation data expressed in XML form.

Figure 5.1 depicts the three levels of interaction with information required by XMPF that have all been catered to by the Orbeon Forms platform:

- The handling of large quantities of data is available through an implementation of the eXist XML database, which supports XPath and XQuery [ORB11].

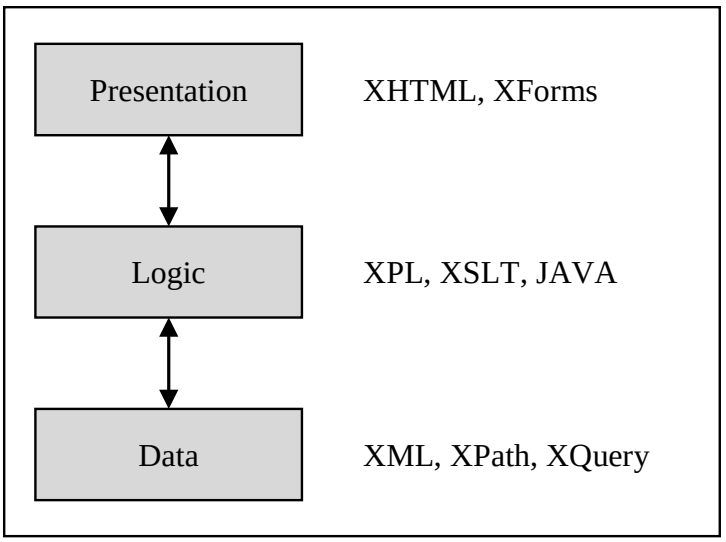


FIGURE 5.1: Capturing, Processing and Presenting Information in Orbeon Forms

- Thanks to the different built-in XPL processors which encapsulate logic to perform generic functions, applicational logic is readily available for accessing databases using SQL, calling WSs and executing XSLT operations. For more advanced tasks, JAVA processors are able to call, compile and execute external JAVA classes.
- The presentation of information is handled by XHTML [XHT10] and XForms [XFO09], which are applications of XML for the specification of user interfaces.

The page flow controller is at the center of every Orbeon Forms web application. It dispatches incoming user requests to individual pages built out of models and views, following the Model-View-Controller (MVC) architecture first thought of by Reenskaug [REE79], and then applied for the first time in Smalltalk-80 by Krasner and Pope [KRG88]. The MVC design pattern divides an application into three categories of components:

1. *Model:*

   The model contains the data represented by the view, and the logic necessary to react to the users manipulations.

2. *View:*

   The view uses the data contained in the model to generate and output a specific representation.

3. *Controller:*

   The controller is the component allowing the user to interact with the model state and thus change the view's representation of it.

Web-applications, like other interactive software systems, can benefit by being architected with the MVC design pattern shown in Figure 5.2, and adaptations for a lot of the major programming languages have been developed [LER01].

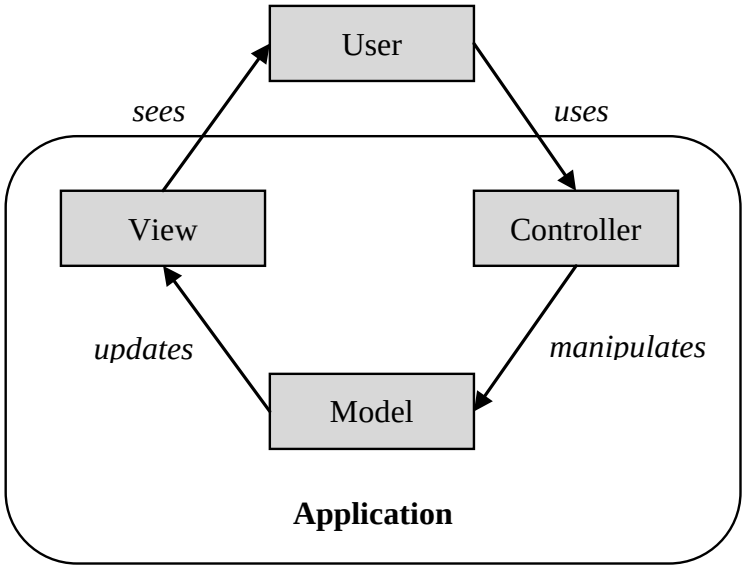


FIGURE 5.2: MVC Architecture

The Orbeon Forms platform can be installed on a multitude of application servers and therefore behaves like any other client-server architecture.

5.2 Web Service Structure

In regards to model pipelines, further structural constants can be identified. Besides the page flow controller, every model pipeline built by the XMPF features a WS schema, a WS description, an XPL pipeline and usually employs a JAVA processor. Their exact interaction is described near the end of this section. Furthermore, several WS types can

be built with the aforementioned files, which will be introduced as the very last topic of discussion.

5.2.1 Page Flow Controller

Each page flow controller has the same basic make-up shown in figure 5.3, which allows to describe the pages offered by a model pipeline and their locations on the server: the XPL pipeline of the WS, its description and the matching schema.

Location on server	MVC category	Functionality provided
/ws-url/	Model	WS Pipeline
/ws-url/wsdl/	View	WS Description
/ws-url/schema/	View	WS Schema

FIGURE 5.3: Structure of the Page Flow Controller for a Model Pipeline WS

The XPL pipeline holds the logic of the WS and thus represents the model of the application, while the WS description and schema both are declared as views to enable the WS to be exposed, because they need to be "viewable" to allow the WS to be recognized as such by external applications.

5.2.2 XML Schema

Employing XML schemata is a common and successful practice to formulate the structural constraints of XML documents. The XMPF uses this approach to isolate the XML structure of the WS's SOAP request and response from the WS description and configuration. Figure 5.4 describes the general structure of an XML Schema found in a XMPF WS.

First, the different XML data types used in a specific WS are declared. Those range from base types, over simple types, to the complex types. The so called base types, are derived from the native XML base types like integers or strings. They allow to restrict

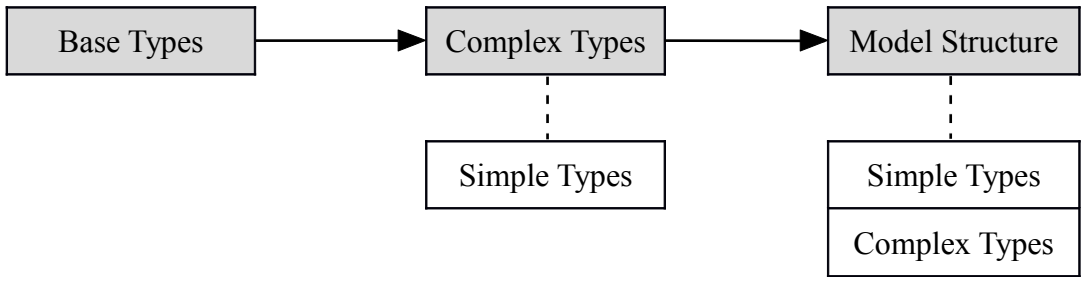


FIGURE 5.4: Structure of the XML Schema for a Model Pipeline WS

the input to certain values, as in only permitting integer values between 0 and 23 to express the hour of the day for example. Next, the complex types are declared. These can be composed of simple types and/or other complex types that may make use of the previously described base types. Finally, the model structure is defined at the end of the XML schema file. It is comprised of its own simple types and the aforementioned complex types. It serves to provide the WS description with the make-up of the SOAP request and response.

5.2.3 Web Service Description

The XMPF employs WSDL to describe the functionality offered by its WS. It allows to instruct external programs how the WS can be called, what parameters it expects, and what data structures it returns. Figure 5.5 schematically shows how such a WDSL file is structured.

The WS description is partitioned as follows:

- *Types:* Provides data types definition used to describe exchanged messages. A XMPF WS schema is imported at this point.
- *Message:* Abstract representation of the data being transmitted through the WS. In the XMPF this refers solely to the unique WS request and WS response.

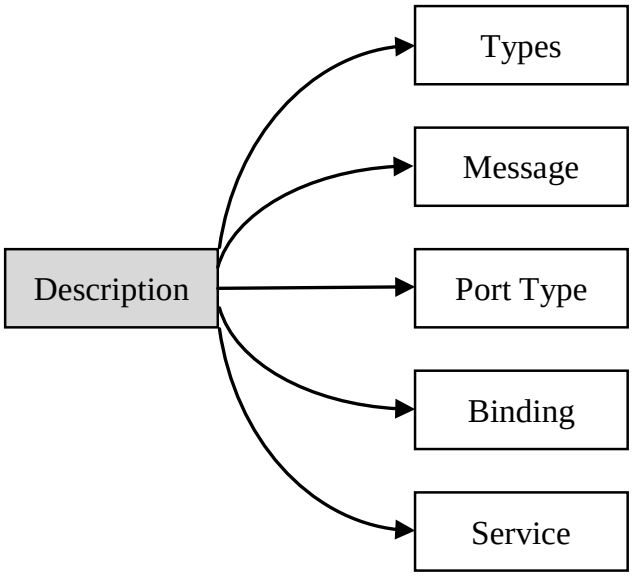


FIGURE 5.5: Structure of a WS Description

- *Port Type:* Usually a set of abstract operations. However, the XMPF only defines only one port type referring the input message (request) to the output message (response).
- *Binding:* Specifies a concrete protocol and data format specification for the operation and messages defined by a particular port type.
- *Service:* Used to aggregate a set of ports which describe an address for a binding, thus defining a single communication endpoint (the WS URL).

5.2.4 XPL Pipeline

The XPL pipeline in a model pipeline WS has two distinct, not mutually exclusive tasks: it gets a SOAP request and either redirects it to a CSP or another WS. This can be done directly or by calling a JAVA class. Afterwards, it transforms the result into a SOAP conform response, and outputs the generated XML.

Figure 5.6 shows the different steps of this procedure, which each utilize distinct Orbeon Forms processors. The generators are part of a special category of processors that have no XML data inputs, only outputs. The request generator is used at the top of an

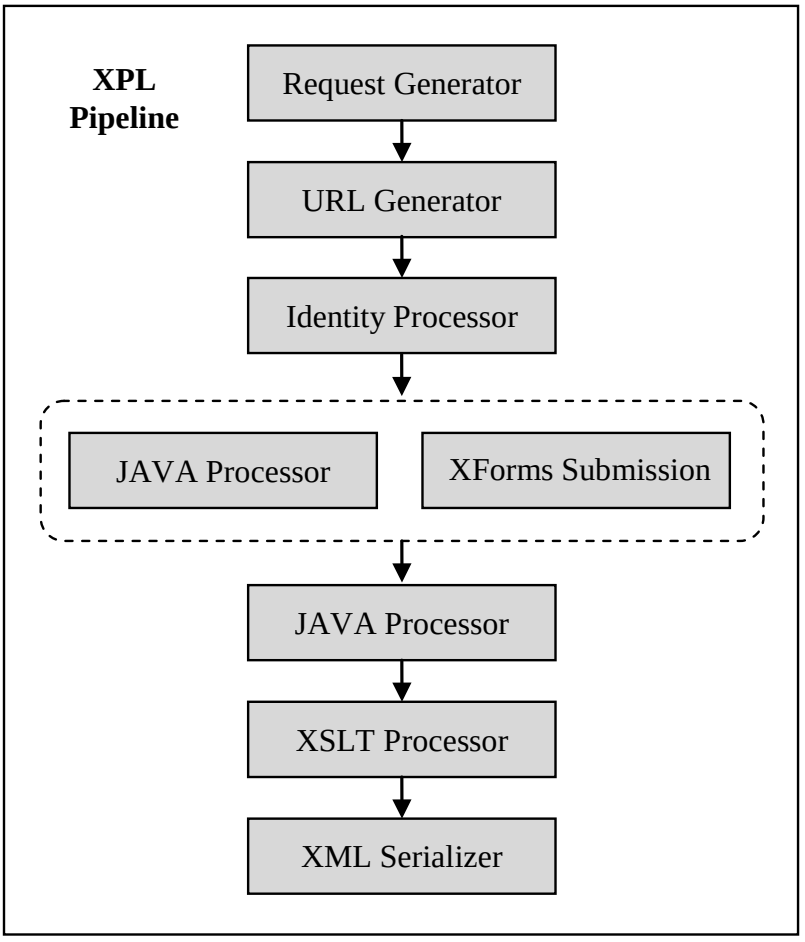


FIGURE 5.6: Structure of the XPL Pipeline for a Model Pipeline WS

XPL pipeline to generate XML data from a non-XML source by streaming XML from the current HTTP request. The URL generator now fetches a document from the temporary URL created by the request generator and produces an XML output document. Afterwards an identity processor is called, which aggregates chosen parts of this XML document to ensure that only relevant data is transferred to the next processor.

The next step in an XPL pipeline offers three options. Either the data is processed and transferred to a CSP in the pipeline itself through the use of the different processors that the Orbeon Forms platform offers. But generally the XML data has to undergo some major transformations and/or interpretation before it can be handled by external simulators. For this reason XPL pipelines call upon custom JAVA processors that enable the user to easily implement operations on that data in a flexible manner. It also allows for a larger range of possibilities to pass the information to CSPs (file transfer,

TCP/UDP data streaming, etc.). Another possibility is calling upon other, possibly non-XMPF implemented, WS to handle the processing. This is done by making a so called XForms submission, which allows to call services like model pipeline WSs from XPL without the need to create dedicated XForms pages, as it normally would be required by Orbeon Forms. Whatever the type of processing opted for, the output of the current processor is then used in an XSLT processor to create a SOAP response conforming the specifications of the WSDL file. The data is then finally handed to the XML serializer. Contrary to generators, serializers do not output XML. Therefore, instead of reading from of URL like the URL generator does, the XML serializer writes its data input as XML into a temporary URL that is exposed as SOAP message.

5.2.5 JAVA Processor

Figure 5.7 represents the basic make-up of the previously mentioned JAVA processors.

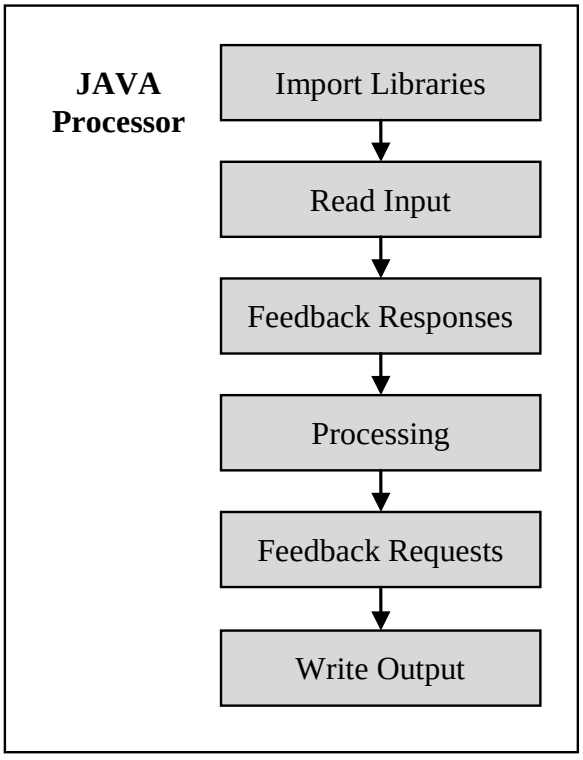


FIGURE 5.7: Structure of the JAVA Processor for a Model Pipeline WS

At first, the necessary libraries needed by the JAVA class for the interaction with Orbeon Forms and XML data are imported. Afterwards the XML input is read and saved in

structured lists of strings. Then, if present, the feedback requests of interest to this specific WS are read and handled, creating the appropriate responses by altering the original processor input. The input is then processed and sent to a model in an external CSP where a simulation is executed according to that information. Once the simulation is done, the data is transferred back to the processor where optional feedback requests to other WS in a model pipeline are generated. The results of the simulation and the feedback requests are finally written to the output of the processor.

5.2.6 Overall Structure

Figure 5.8 is a representation of how all the previously described components work together.

In an XMPF model pipeline, to generate a response to a specific request, the WS interface to an external CSP uses a page flow controller to reserve and specify the URL addresses on the server for all the files needed by the WS. The service discloses the data types and their structure with the help of a WS description and the WS schema it imports. The XPL pipeline accepts the messages sent to the WS and sends them to an actual simulator, most probably through the use of an optional JAVA processor, or invoke a WS. After the external simulation or service has been executed with the conditioned data from the request, an output is produced and sent back to the XPL pipeline, which transforms the received information into a SOAP response adhering to the specifications of the WS description.

5.2.7 Web Service Types

The XMPF feature three distinct types of WS which all make use of this architecture: atomic WS, composite WS and the unique Experiment WS. So called atomic WS are services directly interfacing with an external CSP. They usually make use of the JAVA processor to interpret the XML data and send them in an appropriate format to the external simulator. So called composite WS serve to chain several atomic WS together

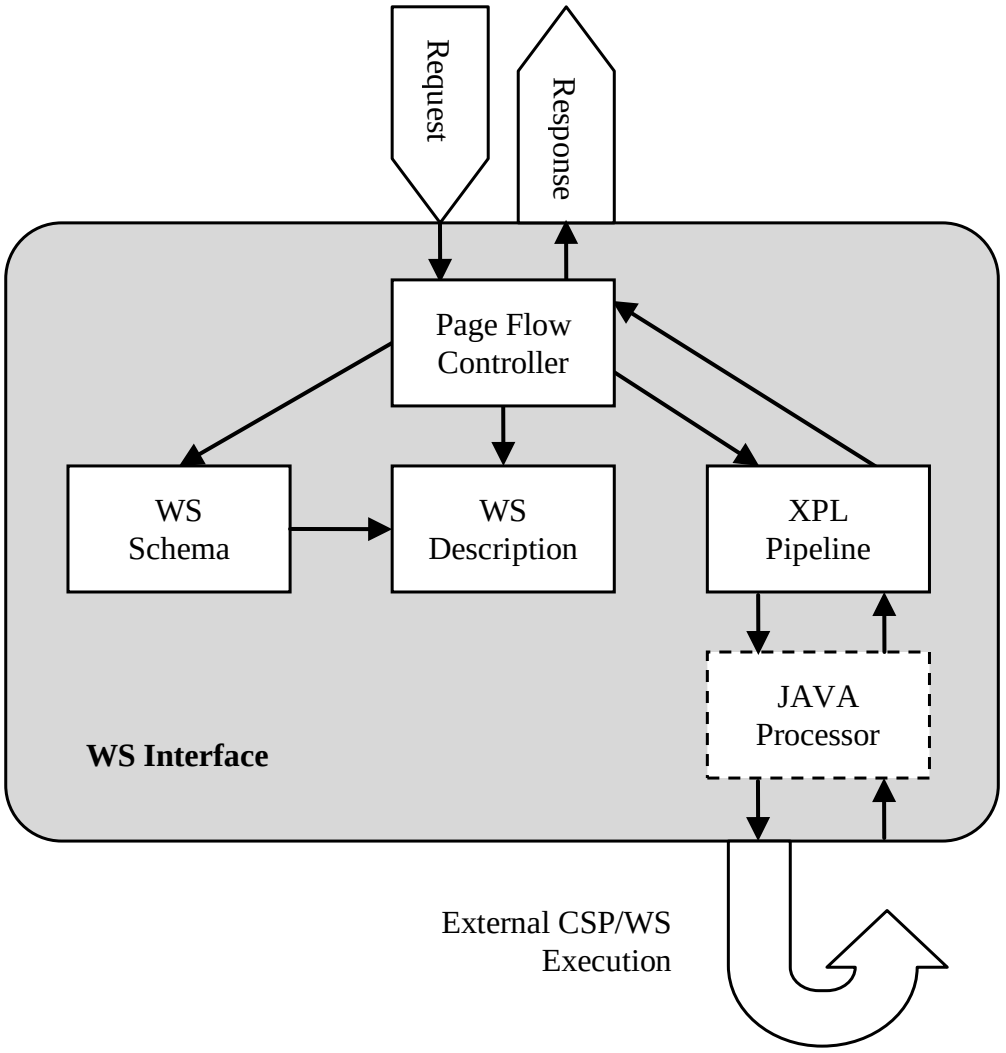


FIGURE 5.8: Overall Structure of a Model Pipeline WS

to create composed simulation. These generally call the other WS directly from their XPL pipeline. The Experiment WS is unique in that it serves the purpose to offer the user a consistent interface to model pipelines. It provides the feedback capabilities of model pipelines, allows the user to run multiple consequent runs of different model pipelines, and implements a way to control simulation runtimes. To provide all this functionality requires the use of a JAVA processor as shown in figure 5.8. Every server running model pipelines is required to have exactly one instance of it running, and those instances are consistently equal across all servers, as the Experiment WS does not implement any simulation specific functionality.

Figure 5.9 schematically demonstrates how the discussed components of a model

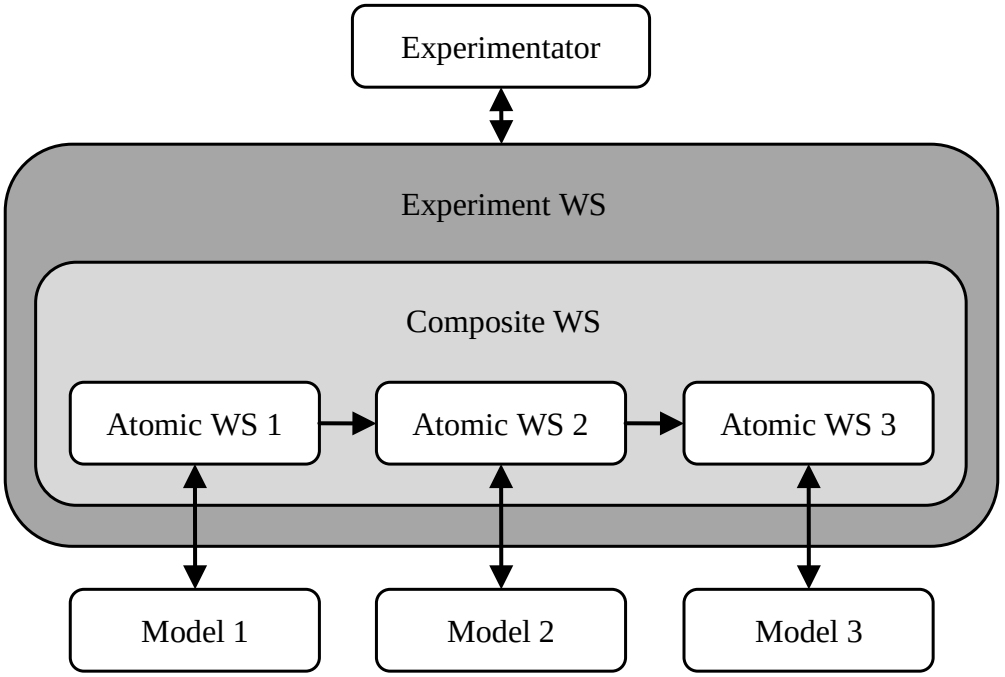


FIGURE 5.9: Model Pipeline Architecture

pipeline achieve to create a distributed simulation. The experimentator calls upon the Experiment WS to start the composite WS of his choice with the appropriate parameters. The composite WS now transfer its input the atomic WSs it incorporates, starting with the atomic WS 1. That WS chooses the information that is relevant to it and passes it on to the model it is interfacing with after conditioning the data. The model creates an output that the atomic WS 1 retrieves and messages its requestor with it. The requestor here being the composite WS now forwards the original input with the newly obtained output to the atomic WS 2. The same procedure is repeated until the composite WS receives an output from the atomic WS 3 and passes the information back to the Experiment WS, which will check if reruns of the composite WS are necessary to process open feedback events, or if the user wanted to execute additional runs with different parameters. After all runs have been completed, the experimentator receives the end results of his simulation.

5.3 Web Interface Structure

The XMPF also offers the possibility to create web-based graphical user interfaces for operators of model pipeline simulations. These are based upon XHTML technology and use the model pipeline simulations like any other third party application would. The framework however is able to automatically build the content of the web interfaces by knowing the structure of the WS they should cater to. This functionality of the XMPF makes the creation of GUIs for distributed simulations as easy and intuitive as the implementation of model pipelines themselves.

A model pipeline web-GUI is built with four constant files: the page flow controller needed by every Orbeon Forms application, a form view to operate the simulation, a result view which shows the generated output, and an XPL pipeline to delegate the data processing to an appropriate WS.

5.3.1 Page Flow Controller

Figure 5.10 demonstrates what the make-up of a standard page flow controller for an XMPF web-GUI looks like.

Location on server	MVC category	Functionality provided
/gui-url/	View	Form View
/gui-url/result/	View	Result View
/gui-url/result/	Model	GUI Pipeline

FIGURE 5.10: Structure of the Page Flow Controller for an XMPF Web-GUI

It features only two different URLs: one where the form view is declared as view for the experimentator to input the starting parameters for the simulation, and one view where the results are shown, while a model handles the processing, following the MVC architecture.

5.3.2 Form View

Figure 5.11 shows the structure all form views have in the XMPF. They have been dubbed this way because they are actually represented as forms for the user to input their simulation parameters.

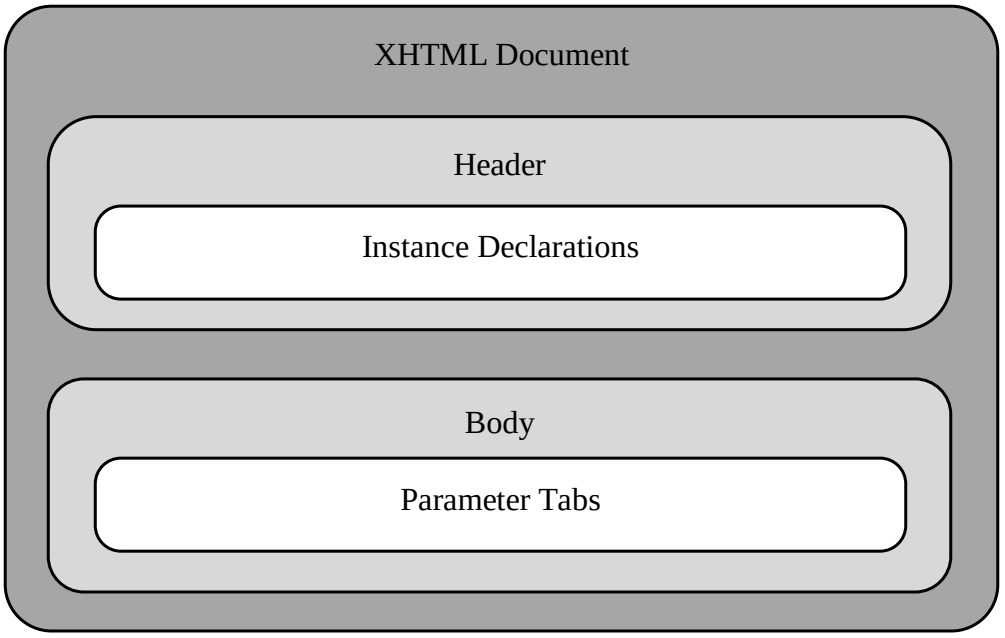


FIGURE 5.11: Structure of the Form View for an XMPF Web-GUI

Like most XHTML documents, it features a header and a body. The header is used for all the so called instance declarations. A complete declaration of a specific experiment is always required, to inform the form view of the general XML structure of the simulation data. Furthermore, all simple and complex types which can occur an indefinite amount of times in the XML construct must also be declared. The body holds the information on how to present the instances declared in the header. For each of the top-level types which build a model (see figure 5.4), the web-GUI is instructed to create a tab which holds all the other types it comprises, as seen in figure 5.12. The instances of recurring types declared in the header are used to give the web-GUI the capability to add and/or remove them to the XML data which builds the experiment.

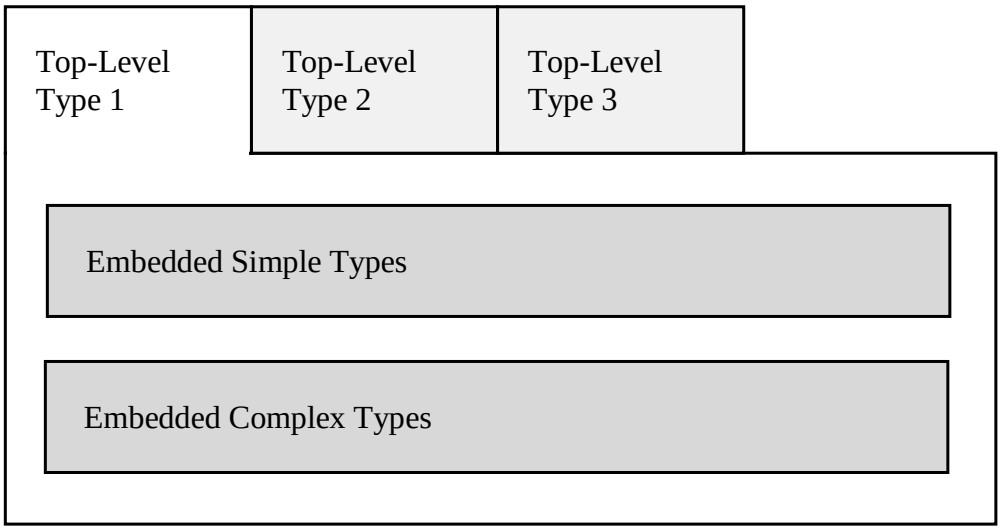


FIGURE 5.12: Layout of a Form View Web Page

5.3.3 Result View

The XHTML document for the result view shown in figure 5.13 is very similar to the form view. It does however not come as a form, since the output is not editable. The other differences here are that there is no need for instance declarations of the XML data structure in the header, since it is not built dynamically by the operator of the model pipeline, but is instead a fixed input to the result view. Furthermore, this web page partitions the information in the body into input, output and XML. For each of those categories a tab is created holding the correspondent information, again in a tabbed manner, except for the XML tab which hold the raw XML message sent by the model pipeline as output.

5.3.4 XPL Pipeline

For the sake of completeness, figure 5.14 shows how the GUI pipeline is made up similarly to figure 5.6. The structure is obviously a lot simpler, as it basically holds only one command: the XForms submission processor to delegate the processing of the simulation data to the model pipeline the web-GUI was implemented for.

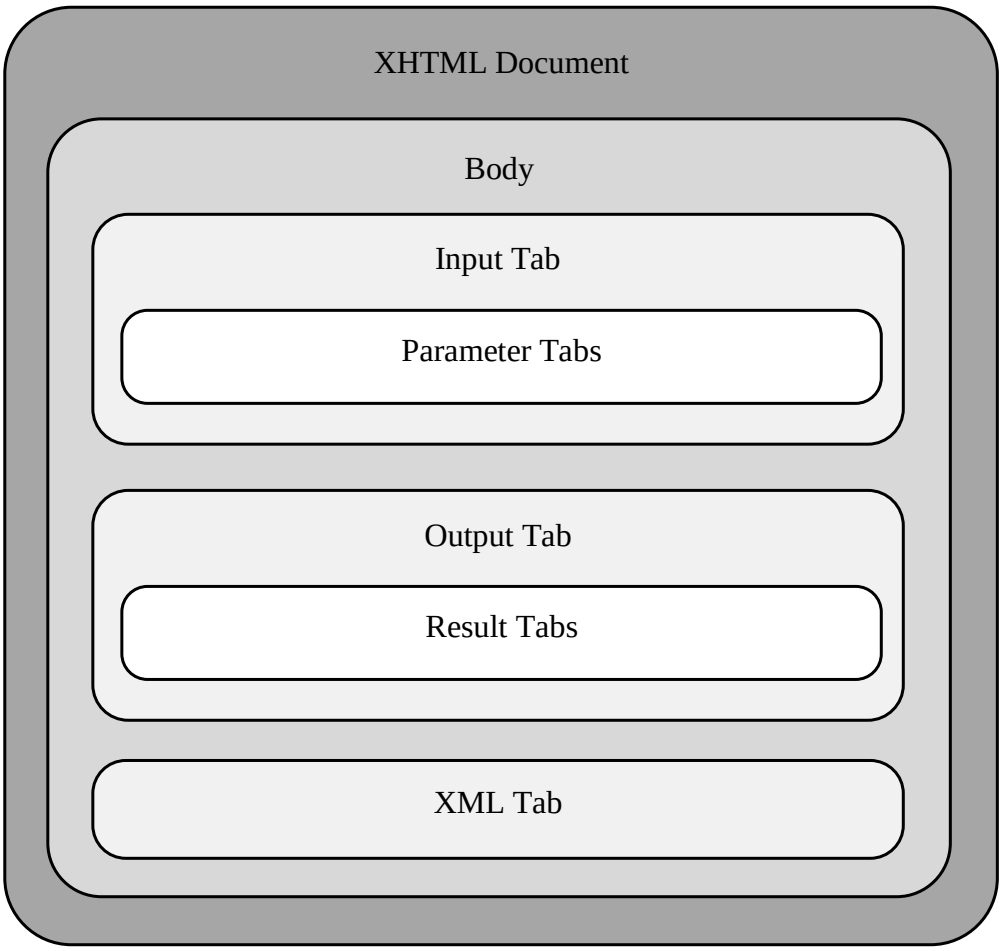


FIGURE 5.13: Structure of the Result View for an XMPF Web-GUI

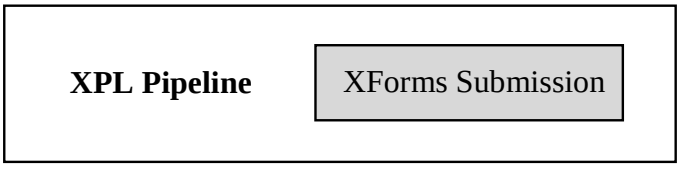


FIGURE 5.14: Structure of the XPL Pipeline for an XMPF Web-GUI

5.3.5 Overall Structure

The overall GUI pipeline structure uses all those files to present the user with an effective way to conduct experiments with any given model pipeline built with the XMPF. As seen in figure 5.15, the setup is pretty straight forward:

1. The web-GUI is called and opens up the form view as starting page.

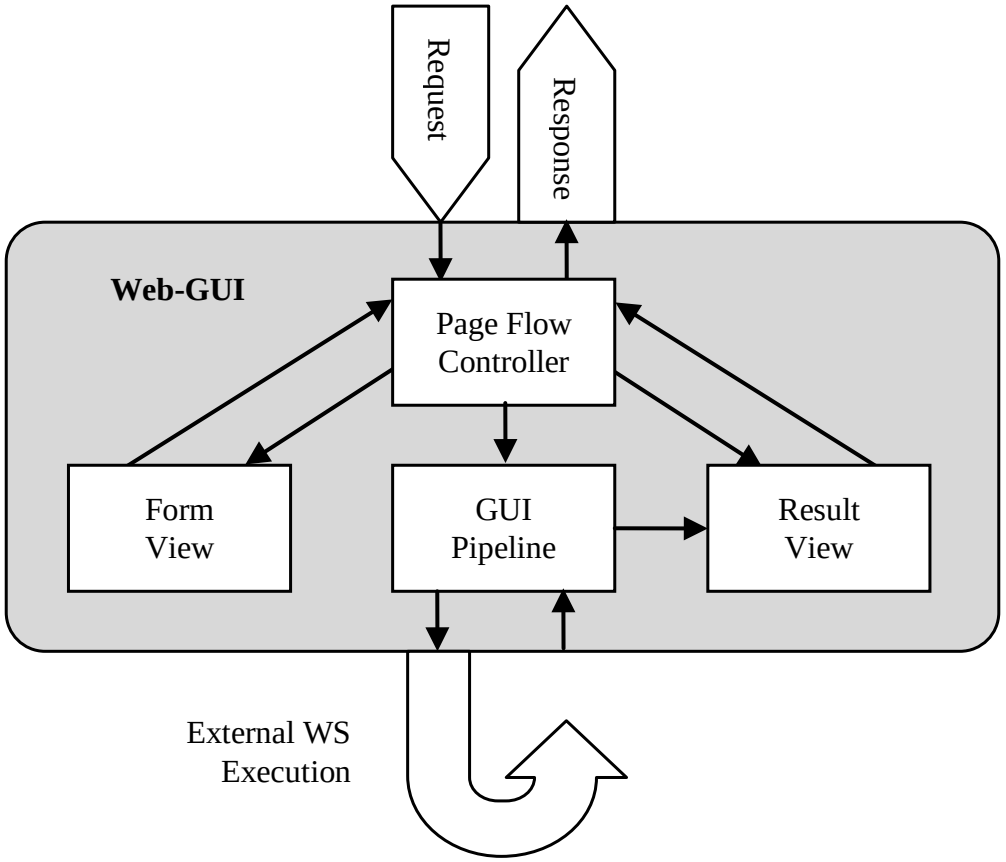


FIGURE 5.15: Overall Structure of a Web-GUI

2. The experimentator enters the input data for the simulation into the form.

3. That input is submitted through the page flow controller to the GUI pipeline, which passes the XML data to the external model pipeline WS.

4. The resulting XML created as response by the model pipeline is then presented in the result view of the web-GUI.

5.4 Framework Description

The XMPF principally is a cooperation of JAVA classes designed to create model pipelines based upon the previously discussed structures and mechanics that rely on the Orbeon Forms Platform to operate. Its functionality is described in this section. JAVA has been chosen as the programming language of choice for this project for its

simple grammar, portability, flexibility and prevalence. Furthermore it is understandably easier to generate JAVA code out of a native JAVA application, which is needed in this framework.

The WS and GUI interfaces for model pipelines have been streamlined to such an extent, that the users need for input concerning their implementation has been severely reduced. Only two different sets of instructions are required to be specified by an operator of the XMPF to design new model pipelines: the data types used in the simulation and the logic in the JAVA processors to communicate the SOAP requests to the actual simulators. Since the application logic will differ greatly between the external CPSs, its implementation cannot be fully automated, and user input will always be required to a certain extent. However, with the specification of the correct types to use in a simulation, everything else can be generated by the framework. The different data types that can be declared within the scope of a model pipeline have already been introduced in figure 5.4. These shall be examined more closely in the following passage.

5.4.1 Base Types

A number of the data types are derived from XML data constructs. Base types are a representation of what is known as simple types in XML [XML00]. They constrain the values that may appear in an XML element. Apart from a name, the XMPF needs a base type to be specified with schema, a base, and a list of attributes, as seen in table 5.1.

The schema currently can only refer to the XSD to make use of the primitive data types it offers, which can be specified in the base attribute. At the moment of writing the base can either be the string, integer or double data type. It is however planned to expand the range of bases available and add the option to specify user generated schemata. The list of attributes each base type requires can have two distinct representations, one for textual bases (string) and one for numerical bases (integer, double). The attribute list for strings is an enumeration of different textual values that restrict the input for a simple type with that base type as base to a specific choice. The attributes for a numerical

Base Type	
Attribute	Value
Name	name of the base type
Schema	XSD
Base	string, integer, double
Attributes	list of attributes for that type

TABLE 5.1: Structure of a Base Type

base type only holds two elements: a minimum and maximum value. A simple type with that base type as base could therefore only hold numerical values between that minimum and maximum.

5.4.2 Simple Types

Simple types are a derivation of what is known as a complex type element in XML. Like base types, these also need a name, schema and base. The schema and base may be from the standard XSD library, but can also use any of the base types previously defined either in this model pipeline, another model pipeline, or even from third party WSs. Furthermore, instead of attributes it is necessary to specify a minimal and maximal number of occurrences, and the use, as seen in table 5.2. The minimal and maximal number of occurrences define how often a simple type may come up in an XML request to the model pipeline. The use clarifies in what circumstances this simple type is used. It is either used in the input parameters, the outputted results, or internally to build complex types which in turn appear in the input, output or internally.

5.4.3 Complex Types

A lot of the attributes used in simple types also make an appearance in complex types shown in table 5.3, which are derived from XML complex types.

Simple Type	
Attribute	Value
Name	name of the simple type
Schema	XSD, user created schema
Base	string, integer, user created data type
Min. Occurrence	minimal amount of times that type may occur
Max. Occurrence	maximal amount of time that type may occur
Use	input, output, internal

TABLE 5.2: Structure of a Simple Type

Complex Type	
Attribute	Value
Name	name of the complex type
Origin	schema that type is defined in
Min. Occurrence	minimal amount of times that type may occur
Max. Occurrence	maximal amount of time that type may occur
Use	input, output, internal
Simple Elements	list of simple types in that type
Complex Elements	list of complex types in that type

TABLE 5.3: Structure of a Complex Type

What changes here, is that a complex type does not rely on a schema and base for its content, but on a list of simple and/or other complex data types that must have been declared beforehand. The origin is the schema that the type originates in, which can be from the model pipeline currently being built in the XMPF, another already available one, or an external WS.

5.4.4 Feedback Events

Specifying feedback requests and responses is only necessary if the XMPF model pipeline being built will feature models interacting with each other, which cannot be sequenced in an appropriate order in the pipeline, and if the pessimistic feedback approach is not desired, as that mechanic does not require to distinguish the concerned simulation data (see chapter 3).

Feedback Request	
Attribute	Value
WS	WS to request a feedback from
Tags	list of concerned tags of the above WS

TABLE 5.4: Structure of a Feedback Request

The structure of feedback requests is shown in table 5.4. The attributes needed are the name of the WS that shall receive the feedback request, and a list of tags of that WS that are concerned by this request. A feedback response, as shown in table 5.5 only requires a tag name for the XML element that may be influenced by a feedback request and thus generate a reaction.

Feedback Response	
Attribute	Value
Tag	name of the tag that is eligible to generate a response

TABLE 5.5: Structure of a Feedback Response

5.4.5 Import Sources

Import sources have to be introduced with their name, the URL of their schema and of their WSDL description, as seen in table 5.6.

They can be used in any model pipeline, but become especially relevant for composite WS. They allow to declare import sources for other XML schemata that are to be used

Import Sources	
Attribute	Value
Name	name of the base type
Schema	URL of the WS schema file to import
Service	URL of the WS WSDL file to import

TABLE 5.6: Structure of an Import Source

in the current model pipeline, so that data types do not have to be redundantly specified. Since composite WSs use a combination of atomic and/or other composite WSs, the input and output data types of the overall chain of WSs will already have been declared at some point in the WSs directly interfacing with the external models, and declaring them again in each new composite WS is unnecessary.

5.4.6 Automated Construction

The XMPF saves the details entered by the user concerning the different data types in a dedicated class for it. This class is implemented as a singleton, which is a design pattern that restricts the instantiation of that class to exactly one object. Thus, the same information is made available to the rest of the framework, which task it is to generate model pipelines and their GUIs, in a uniform and consistent manner.

Figure 5.16 shows a rough diagram of the XMPF architecture's implementation. As a singleton, only one instance of the data class which is holding all the user defined data types can be created. That object is then used by the packages for the creation of the Experiment WS, model pipelines and Web-GUIs. Each of those packages implement classes to create the contents for the individual files these categories require, which have been analyzed previously:

1. *Experiment:*
   Specialized model pipeline page flow controller, WS schema, WS description, XPL pipeline and JAVA processor.

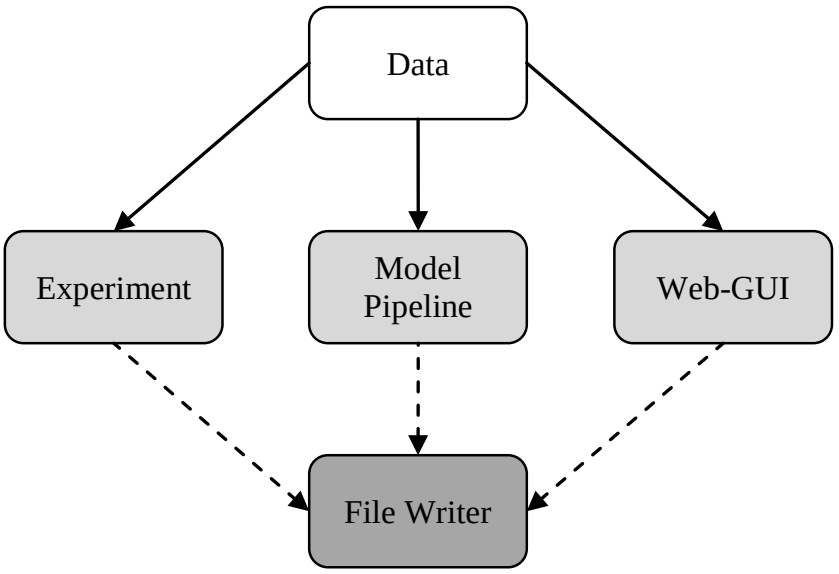


FIGURE 5.16: Architecture of the XMPF Implementation

2. *Model Pipeline:*

   Generic model pipeline page flow controller, WS schema, WS description, XPL pipeline and JAVA processor for atomic WSs.

3. *Web-GUI:*

   Generic web-GUI page flow controller, GUI pipeline, form view and result view.

Each one of these classes uses fairly similar algorithms to iterate over the data types held by the data class object to generate XML, XSD, XPL, WSDL and JAVA program code in a string format. A very simplified example of this procedure is shown in figure 5.16 in the form of pseudo-code. The procedure iterates over all the complex data types entered in the data class instance by an operator of the framework. For each simple data type included in that complex data type, the appropriate code is generated. For each complex data type included in that complex data type, the same procedure is called recursively, until the complex data types present are only built upon simple data types. The contents generated in the classes of those distinct packages is then written to actual, usable files, using a file writer class. The files created in that fashion can then be used on any server running the Orbeon Forms Platform to function as model pipelines or their GUIs. The interested reader may find it useful to take a look at the

UML (Unified Modeling Language) diagrams of the classes that make up the XMPF presented in appendix B [UML12].

```
PROCEDURE create code for all complex data types
FOR each complex data type in the data class object
    FOR each simple data type in that complex data type
        create appropriate code
        FOR each complex data type in that complex data type
            CALL PROCEDURE create code for all complex data types
```

CODE 5.1: Code Creation for Complex Data Types in the XMPF

The only element missing that currently still requires a programmers manual intervention, is a relatively small part of the JAVA processor, that communicates the XML data to the external CSP and transforms the information back into XML after the CSP's execution. It is however technically possible and therefore planned to streamline that process by using modules for established simulators in the framework that would be able to quasi-automate that process.

5.5 Development Tool

Since one of the main objectives of the XMPF is to provide a way to create distributed simulations in heterogeneous environments in an especially simple and user-friendly way, a prototypical GUI for the framework has been implemented to even further improve upon the ease-of-use, which will be presented in the following.

Figure 5.17 shows the main and currently only window of the XMPF development GUI. The menu bar has two entries:

1. *File:*

   Allows to start, save and load a model pipeline project. Projects are saved as XML files adhering to an XMPF specific schema.

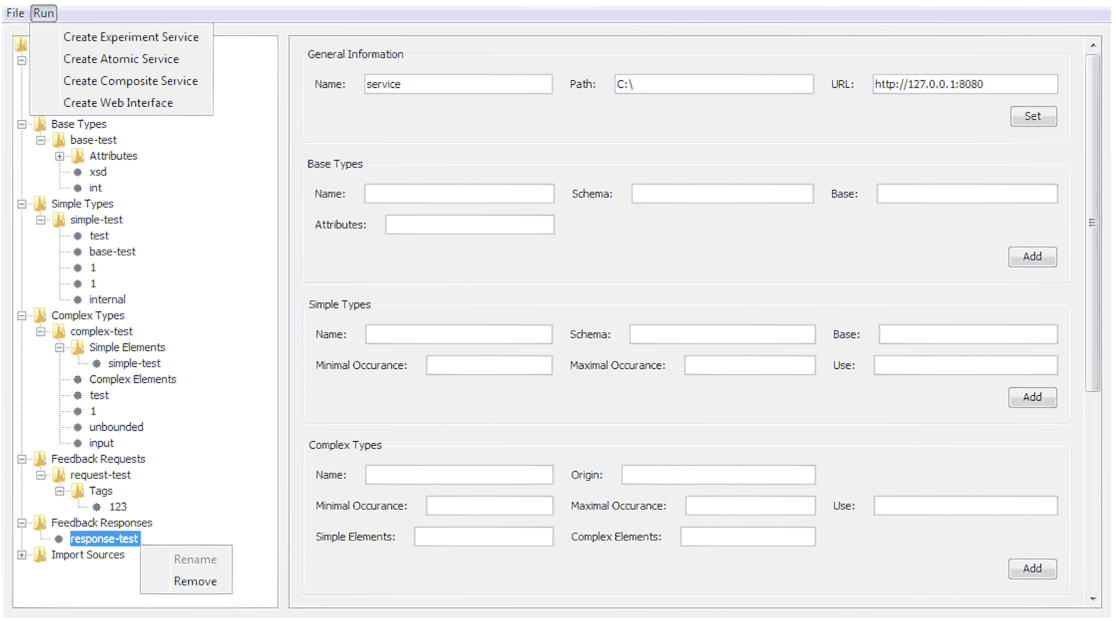


FIGURE 5.17: XMPF IDE

2. *Run:*

   Allows the user to create different kinds of XMPF specific services according to the entries made to the program. Possible choices are the experiment, atomic and composite service, as well as the web interface, which have all been previously discussed.

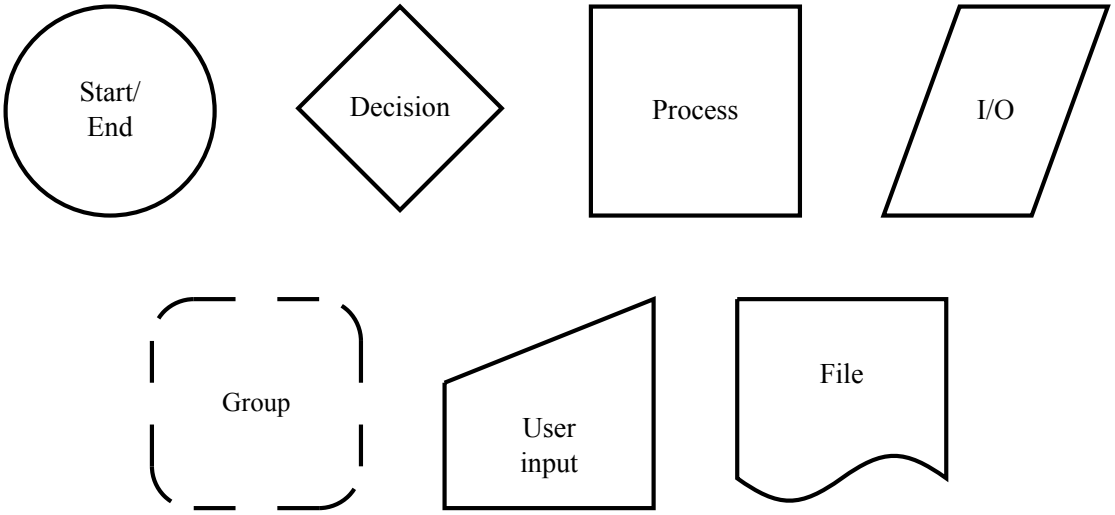


FIGURE 5.18: Flow Diagram Legend

The main work area is divided into two parts:

1. *Input form:*

   Provides input options for all the data types presented previously, that are required by the framework to build a model pipeline.

2. *Data tree:*

   Provides a visual representation of the data types that make up the currently designed model pipeline.

Changes to the entered data types can be done by navigating the tree listing them, and selecting either the "rename" or "remove" option from the context menu. The menu is implemented in such a way that each entry only lists valid menu options, e.g. a simple type needs a name and thus that specific node would not be eligible for removal.

Figure 5.19 is a flow diagram describing how to use the XMPF development GUI by using the elements shown in figure 5.18. At first, the user has a choice of either starting with a new project, or opening a saved one. While a fresh project requires the general information to be entered and new data types to be defined first, opening a previous project allows the user to directly jump into editing the data tree. Afterwards, the new project or edited old project should be saved. That concludes the data input phase of the process. Then come the file creation phase, where the user has to choose what to do with the entered information. If no Experiment WS has been created until now, this has to be done first, as model pipelines require one for feedback handling and automated sequential runs. After that, one can choose whether to create an atomic or composite WS, and an according web-GUI if desired. For simplicity's sake, the diagram omits the writing of the involved files. But this process is directly linked to the base capabilities of the XMPF and was thus explained in the previous passage. Lastly, the user has to decide whether he wants to continue working with the development interface, or exit the program.

The prototype does not yet implement a view of the actually produced program code when creating services or interfaces. However it is planned to add such a feature, especially for the ability to save manual changes the user may want to make to the code. These are after all still necessary to the JAVA processors, as already noted, to create a functioning interface to the external CSPs within most atomic WS.

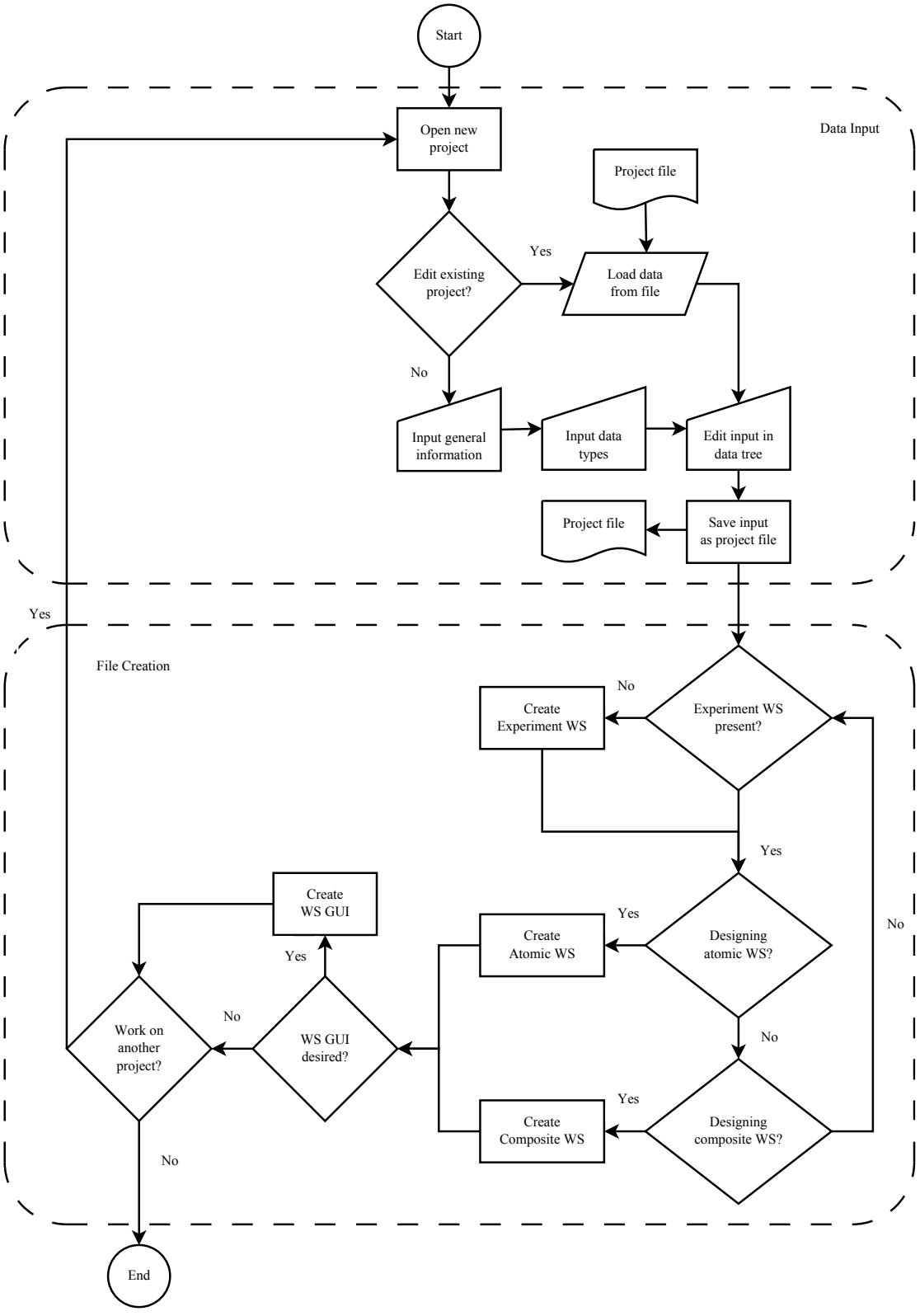


FIGURE 5.19: Flow Diagram for the Usage of the XMPF Development GUI

Chapter 6

Transferring a HLA Simulation to Model Pipelines

6.1 The Dry Port Federation

This chapter aims at comparing model pipelines to the HLA, being the current state-of-the-art of distributed simulation, to demonstrate the differences, advantages and drawbacks. The Sushi Restaurant federation is one of the most advanced and complete examples to demonstrate the capabilities of the HLA, since it uses a large number of the services the HLA provides [MWK12]. Kuhl et al. have authored an in-depth description of this federation [KWD99] and others have had use for it in their research [PER05], [HRX12].

Although the Sushi Restaurant example is very suitable for teaching purposes, its practical relevance is more questionable. One of the many projects handled by the Computer Engineering Research Group of the University of Hamburg does however show that practical relevance, while being adaptable to the Restaurant federation example. This refers to the subject of dry ports.

6.1.1 Basic Structure

The project "Container & Underground" (C&U) has the aim to relieve the existing transport infrastructure of the Hamburg port, while improving the productivity to match the projected future increase in cargo turnover. The creation of dry ports in the hinterland is a sensible strategy to maintain the performance of terminals in the port itself, if they have only limited expansion capabilities [ROL10]. Efficient simulation algorithms are required for a controlled planning and faster implementation of that idea. Therefore, a basic HLA simulation has been developed for this subject, based on the Production - Transport - Consumption plot present in the Sushi Restaurant federation. A complete rewrite of the HLA-coupling was however not necessary, since the basic processes and structures have proven to be fairly similar. The original example was thus only adapted and extended into the new Dry Port federation presented in the following.

An analysis of the Dry Port concept has helped to identify three basic type of events. First, the arrival of goods in form of containers at the port. Secondly, the transport of those containers to the depot. Thirdly, the storing of goods in a dry port depot. Consequently, to simulate a dry port, there must be a Harbor model simulating the arrival of new goods at the port, a Transport model that represents the transport of containers, and a Warehouse model that computes the receipt and storage of goods, as depicted schematically in figure 6.1.

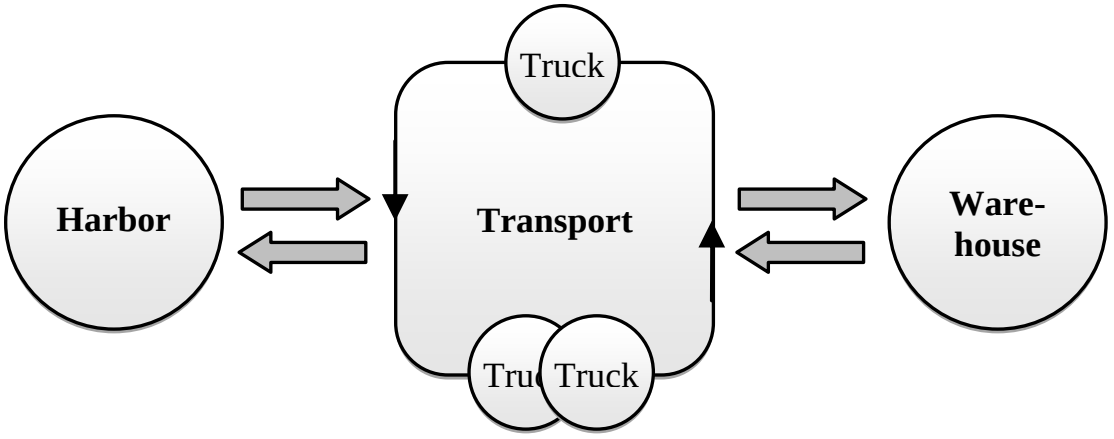


FIGURE 6.1: Schematic Structure of the Dry Port Simulation

6.1.2 Modeled Processes

Similarly to the Dry Port, the Sushi Restaurant federation also has three basic events of importance: the creation of sushi servings by the cook (production), the transport of those servings on a transport belt (transport), and the consumption of said servings by customers (consumption). Due to this structural similarity, the Dry Port concept is easily applicable on the Sushi Restaurant federation example. The models making up the federates only have to be adapted to the operations of a dry port:

- The Production model must be adapted to create containers with goods instead of sushi servings.
- The Transport model has to be changed to transport containers on trucks driving on a road instead of servings on plates riding on a transport belt.
- The Consumption model has to store containers in a warehouse instead of having the sushi getting eaten by customers.

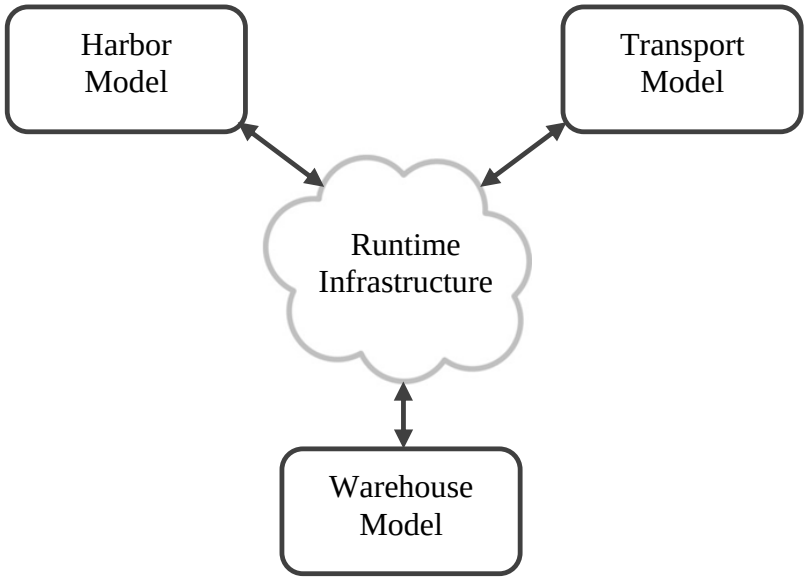


FIGURE 6.2: The Dry Port Federation

Figure 6.2 shows how the Dry Port federation after the conversion. However, due to some technical characteristics of the Dry Port concept, certain aspects of the original

Sushi Restaurant example have to be extended. First, the quantitative termination condition of the restaurant concept, i.e. the federation execution ending after a certain number of servings, makes little sense in a dry port. This requirement has to be converted into a temporal scheduling so that the simulation ends after a specifiable time frame. Additionally, in the Sushi Restaurant federation, no queues where considered in what makes up the Transport model in the Dry Port federation. This topic is however of great importance here to represent reality in a correct fashion. A plate might travel on the transport belt for an indefinite time until it is used for a serving of sushi, or a serving is removed from it, since it comes at no additional costs. However, a truck will not travel to the warehouse without a load, or to the harbor without one, but instead wait until being loaded with cargo or unloaded respectively.

6.1.3 Federation Configuration

Since the HLA uses the RTI as a medium for federates to interact, two additional channels of communications have to be created in regards to the queuing problem, so that the Harbor and Warehouse models can signal the Transport model whether a truck is free to travel or not. The best option within the HLA is with the use of interactions. Both the Harbor and Warehouse models have therefore a method to check with each step in the simulation if a truck is within reach. If that is the case, a specific interaction is sent to the RTI, to which the Transport model is subscribed. This interaction is then evaluated by either the Harbor or Warehouse model. Encapsulated within the interaction is a set of parameters which allow the Harbor model to clearly identify a truck that is currently in the vicinity of the harbor or warehouse. Thereafter, its status is checked. If the truck is unloaded and at the harbor, and the harbor has no new cargo ready for transportation, the truck is queued there. Similarly, the truck is queued at the warehouse, if the truck has cargo and is at the warehouse, but the warehouse has invested all available capacity to unloading other trucks. If the Harbor model has new cargo available, or the Warehouse model ready to unload a new truck, the first truck of the local queue is handled accordingly. Figure 6.3 illustrates the process. The algorithms for the Harbor and

Warehouse models are slightly less complex, but each feature two processes running parallely. Figures 6.4 and 6.5 depict their respective functionalities.

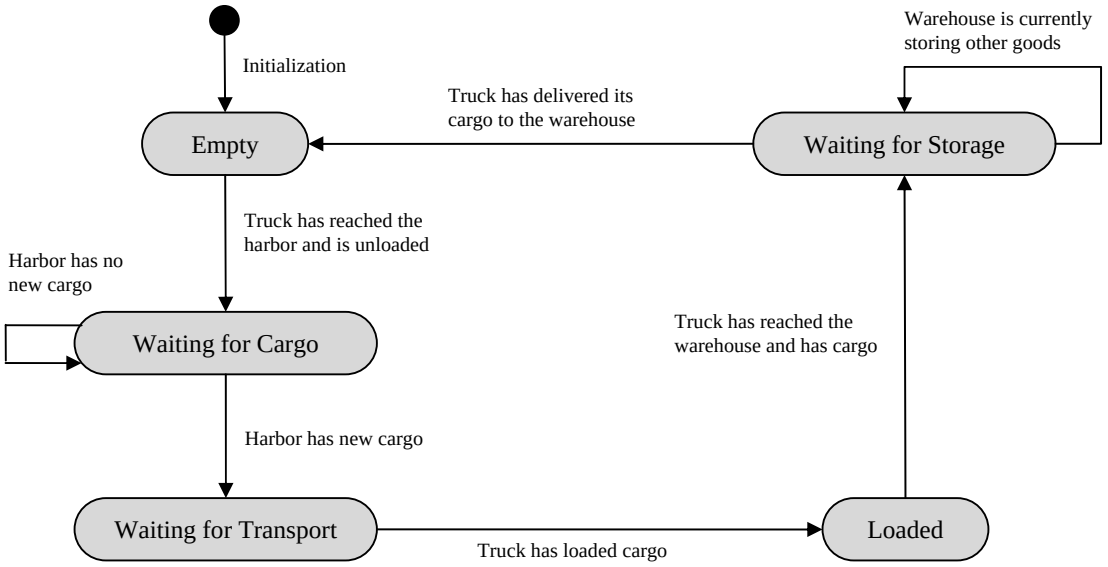


FIGURE 6.3: State Transition Diagram for the Transport Model

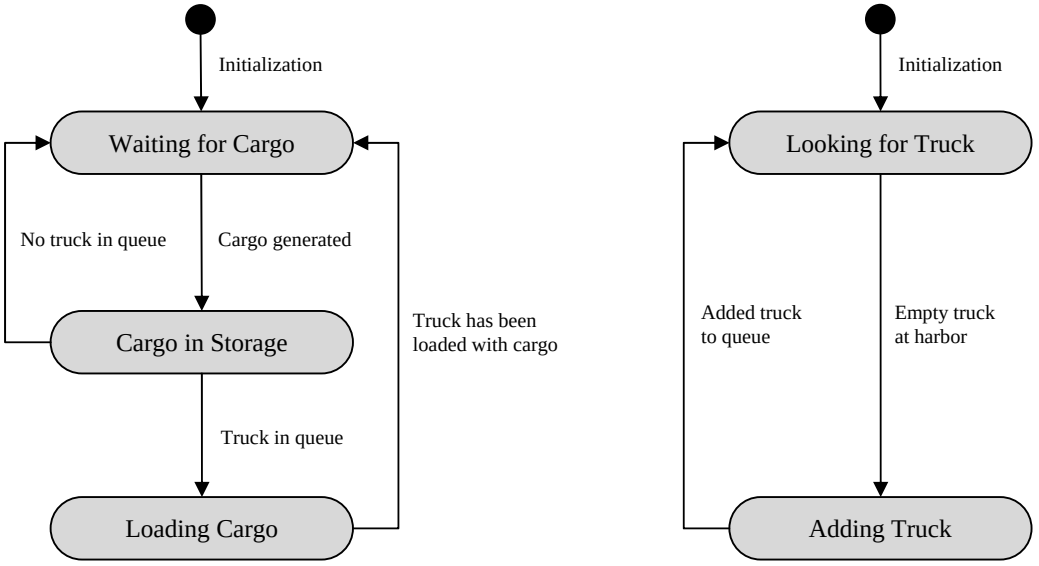


FIGURE 6.4: State Transition Diagram for the Harbor Model

To analyze the inner workings of the Dry Port federation, the presented state flow diagrams shall be summarized in a step by step format hereafter:

1. All federate initialize at the same time. This means for the Harbor federate that it starts generating new cargo and listening for empty trucks, while the Transport

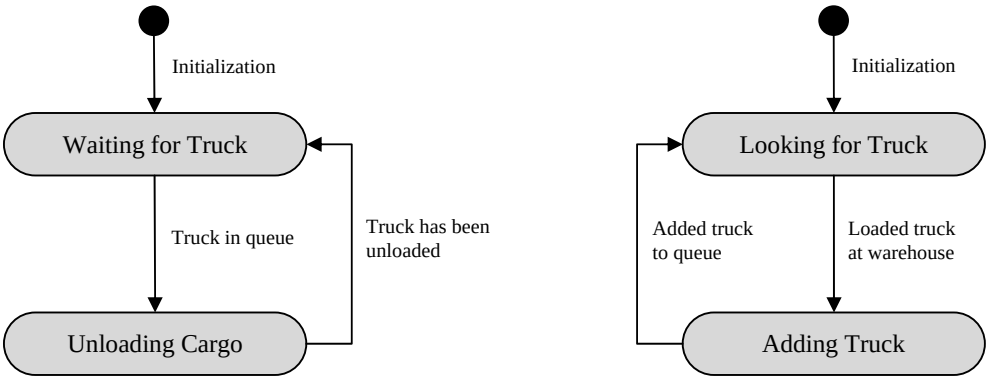


FIGURE 6.5: State Transition Diagram for the Warehouse Model

federate is loading the trucks in the system, starting them at the harbor, and the Warehouse federate is listening for loaded trucks.

2. The Harbor federate generates new cargo after a specific time interval. It checks for empty trucks in its queue and loads the cargo on it as soon as it finds one. Internally, this means for the RTI that a new cargo object instance is registered by the Harbor federate and the ownership of that instance is then transferred to the Transport federate.

3. The Transport federate then moves the truck with a certain speed along a specific distance from the harbor to the warehouse.

4. The Warehouse federate listens for arriving trucks with cargo and tries to unload them. Since the unloading process consumes a set amount of time, every loaded truck arriving during that process will be placed into a queue that will be processed over time. Each time a truck is processed, the RTI transfers the ownership of an according cargo instance to the Warehouse federate, which will delete the instance once handled.

5. After processing in the Warehouse federate, a truck is then moved once again by the Transport federate, this time from the warehouse back to the harbor.

All these steps are actually running simultaneously during the federation execution. The HLA keeps them synchronized and they are using the previously interactions explained in the next section to interact over the RTI.

```
# dryport_1.props:
# properties for dry port federation

Federation.initialTime=0.0
Federation.timeToDo=240
...
Federation.Cargo.numberOfTypes=3
Federation.Cargo.Name.0=Container_100
Federation.Cargo.Name.1=Container_200
Federation.Cargo.Name.2=Container_500
Federation.Cargo.meanManufactureTime.0=25.0
Federation.Cargo.meanManufactureTime.1=30.0
Federation.Cargo.meanManufactureTime.2=50.0
Federation.Cargo.produceSize.0=100
Federation.Cargo.produceSize.1=200
Federation.Cargo.produceSize.2=500
Federation.Cargo.warehousingTime.0=40.0
Federation.Cargo.warehousingTime.1=30.0
Federation.Cargo.warehousingTime.2=45.0
...
Transport.numberOfTrucks=4
Transport.distanceUnit=100
Transport.Trucks.rate=3.0
...
```

CODE 6.1: Excerpt of the Dry Port Federation Properties File

It is worth mentioning that, using the Sushi Restaurant federation as template, the Dry Port federation makes use of a properties file to define several parameters of the execution. Those include the aforementioned simulation runtime, the distance the trucks have to cover and their speed. But other values have been specified here as well: cargo types, sizes, their generation intervals and handling times for example. An excerpt of the properties file is shown in the code snippet 1. This obviously influences how the federation runs and the produced output values.

6.2 HLA Federation Execution

The various simulations making up the federates now in existence are assumed to have been modeled by distinct parties. For this reason, the programming paradigms of those simulations may differ greatly, and their execution is assumed to take place from different geographical locations to leave them in control of the original creators. It is furthermore assumed that they are HLA-compliant and that there is little need for modifications for them to act as federates. The HLA allows to combine those simulations into a combined federation, acting as an interface to mask the disparate model structures.

6.2.1 Federation Management

The first step in managing a federation, is to create an execution associated with a FOM. The FOM of the Dry Port example will have two classes of importance that serve as template for objects: Container and Truck. Classes like Harbor and Warehouse have been implemented internally in the models, but are of little interest to the federation as a whole, since they play no role in the interaction of the federates. Objects are defined by Kuhl et al. as simulated entities of interest to more than one federate that persist for some interval of simulated time [KWD99]. Container is designed to represent and create several pieces of cargo for the federation, that can be manipulated by all its federates. Trucks are also an interaction medium for all the federates, as they represent the mode of transport for the Container objects. There is no need to declare classes for Harbor and Warehouse in the FOM because none of these objects interact in any way with other federates, although they may be modeled internally in the appropriate simulations. The other form of data in the FOM are so called interactions, which represent simulated occurrences. The only interaction in this example is a command to end the federation execution when a predetermined amount of simulation time has passed.

6.2.2 Federate Interactions

The federates then join the federation, which in turn is then defined to the RTI, that is required to run before a federation can be executed. To share interactions during an execution, a federate sends an interaction, The RTI delivers it to subscribing federates, which then receive it. The procedure is slightly more complex for objects. First, a federate has to register a new instance of an object class. The RTI is now aware that a new instance is entering the federation execution. It informs the federates subscribed to certain attributes of that object class, so that they may discover the instance. As the registering federate updates the values of the instance, subscribing federates are able to reflect those values. The registering federate may also remove the instance from the execution. The RTI then deletes the instance and subscribing federates are ordered to remove it.

This approach of publishing and subscribing federates is illustrated in figure 6.6. While the federates A and B publish different classes of objects, all three federates are subscribed to some of them. The most important feature of the HLA in this scenario is that the federate C is unaware of the federates that register the objects it discovers. Much in the same way that the federates A and B are unaware of the federates that discover the objects they register.

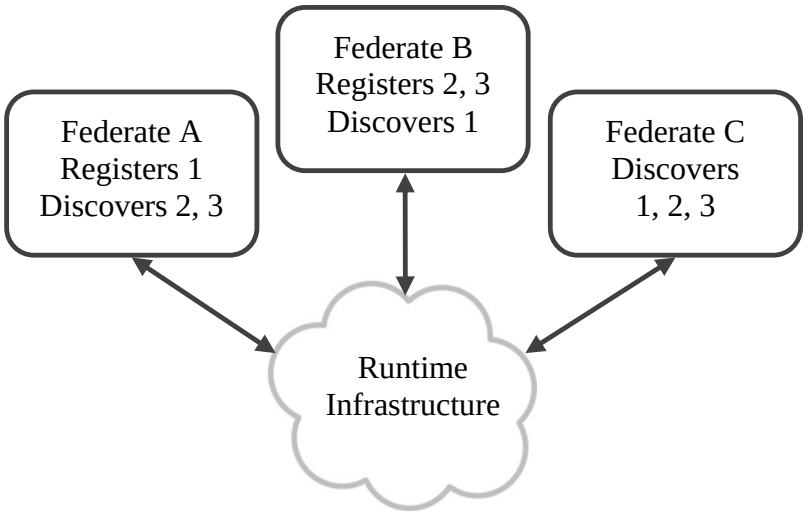


FIGURE 6.6: Publish and Subscribe (confer of [KWD99])

The next important mechanism of the HLA to analyze, now that the publish and discover idea has been explained, is that of ownership and its transfer. In the restaurant federation, it plays a role during the modeling of Container entities, since their attributes are updated in all federates. The change in ownership of such a Container instance is depicted in figure 6.7. The instance attributes are shown on the left, and the life of a container instance progresses from left to right.

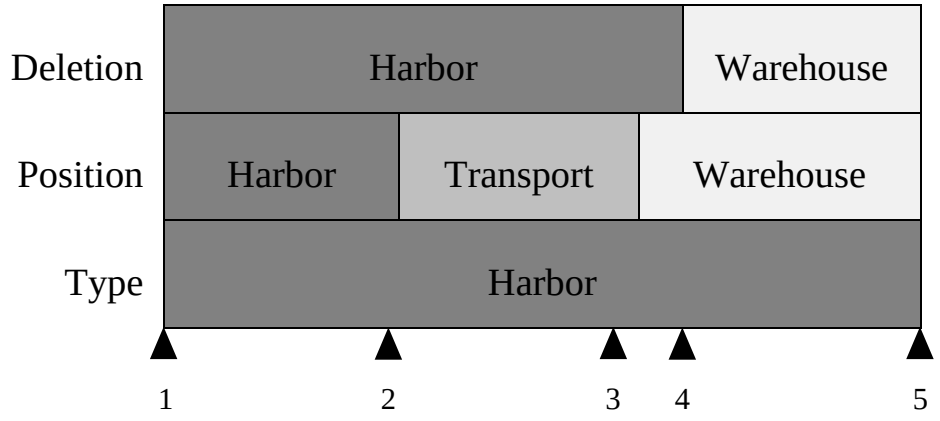


FIGURE 6.7: Ownership Evolution (excerpt of [KWD99])

The different occurrences that are of significance to the concept are numbered below the diagram. Their meaning is listed below:

1. Container instance is registered.
2. Container instance is loaded onto a truck.
3. Container instance is taken from the truck.
4. Container instance is stored away.
5. Container instance is deleted.

The production federate is responsible for modeling the Container instances and as such retains the initial ownership of the instance attributes. The ownership of the position attribute that tracks the instances movement on the transport route is passed to the Transport federate as soon as the instance is loaded onto a truck. When a Container instance

is delivered by a truck, the ownership of the position attribute is passed on to the Warehouse federate. Also given to the Warehouse federate, is the ownership of the deletion privilege when an instance is stored away and thus should be removed from the simulation run. The type of a Container instance is always owned by the Harbor federate, since that attribute does not need updates after the instance's registration.

6.2.3 Synchronization

The last relevant topic for the overview of HLA federation executions presented here is synchronization and time management in general. The HLA ensures that events of a federation execution are delivered to the federates in the correct order. This is a challenge considering the assumed various design paradigms of the models, the computers where the models are running from and their quite possibly differing performances, and their distribution across a network. Unregulated, the order in which events arrive at federates thus could not be guaranteed if all the federates simulated their events in real time. That would disrupt the cause and effect order, and executions of a same federation could produce different outputs. Time managements services in the HLA disregard real time or wall clock time, but instead manipulate the logical time also known as simulation time. As defined by Kuhl et al. [KWD99]:

"Time management services coordinate the advance of logical time within the federation and the delivery of time stamped data."

While the Dry Port federation is conservatively synchronized, the time management services of the HLA support a variety of time management schemes.

- *No explicit time management.*
- *Conservative synchronization:*
  No federate can advance the simulation time except when it is guaranteed not to receive past events.
- *Optimistic synchronization:*
  Federates can compute future events, but receiving past events causes a roll back to an earlier state.

- *Activity scan:*

  Federates go through a message exchange phase until they all agree to advance the simulation time.

However, with the HLA there is a problem remaining regarding coordination. The architecture does not have any specific initialization policies to accommodate the joining and resignation of federates throughout the federation execution. There is no control over the order in which federates join, initialize and finally begin advancing time. For conservatively synchronized federations it is therefore easier to introduce a phased initialization, to guarantee that all federates have joined before any of them begins to advance time. The phases for the example of the Dry Port federation have been defined as follows:

1. *Preliminaries:* All federates join, set time switches and perform publications and subscriptions.
2. *Populating:* Each federate registers their initial object instances.
3. *Running:* The federates now start advancing time.
4. *Post-processing:* Time advancement is stopped and the federates are given time to finish processing any open messages.
5. *Resigning:* All the federates resign from the federation so it can shut down.

6.3 Dry Port Model Pipeline

The previously outlined Dry Port HLA federation now needs to be transposed into a model pipeline simulation. To this end the operational data is analyzed to extrapolate the information relevant to the transfer. Afterwards, the process of building a matching pipeline is described. Finally, the execution of the resulting simulation is studied.

6.3.1 Data Analysis

In order to realize the Dry Port federation as a model pipeline simulation, the input and output variables, the interactions between the distinct models as well as their specific mode of operation have to be identified. That information can be extracted from the source code and the aforementioned properties file of the HLA federation. The results of the analysis of those documents can be found in the tables 6.1, 6.2 and 6.3.

Harbor model
Input variables
Container types
Generation times for all container types
Output variables
Container generation time stamps
Interactions
Ownership transfer request of container instances to the Transport model
Messages the Transport model that truck instances are in reach to enable queuing for empty trucks
Operation
Generation of container types at time specified time intervals
No new containers can be generated while there is still cargo present at the harbor
No trucks can be loaded without containers at the harbor or during container generation
Empty trucks en route to the harbor are queued here
Trucks from the queue are loaded with cargo before leaving for the warehouse

TABLE 6.1: Analysis of the Harbor Model

Transport model
Input variables
Trucks
Container instances from the Harbor model
Output variables
Departure and arrival times of trucks at the harbor or warehouse
Container type loaded on the trucks
Interactions
Acceptance of ownership transfer of container instances from the Harbor model
Listens for messages from the Harbor model that truck instances are in reach of the harbor
Listens for messages from the Warehouse model that truck instances are in reach of the warehouse
Operation
Container transport
Only one container instance per truck

TABLE 6.2: Analysis of the Transport Model

6.3.2 Model Pipeline Design

With the help of this data, the goal is now to create the Harbor, Transport and Warehouse atomic WSs to act as interface for each of the models respectively, as well as a Dry Port composite WS to couple them into a distributed simulation executed by the already presented Experiment WS. The actual models themselves are the same ones that are in use in the Dry Port federation, although they have been stripped down of all their HLA functionalities. For simplicity's sake, these have been implemented directly into the WSs as JAVA processors, but act the same way models of external CSPs would. Since the expected input and output data of the original federation is known, the XMPF can be used to create all the necessary files for the model pipelines. All in all, six different

Warehouse model
Input variables
Processing times for all container types
Container instances from the Transport model
Output variables
Container storing time stamps
Interactions
Acceptance of ownership transfer of container instances from the Transport model
Messages the Transport model that truck instances are in reach to enable queuing for loaded trucks
Operation
Storing of a container for the specified time interval, when a truck from the queue can be processed
No trucks can be unloaded while a container is being stored away
Loaded trucks en route to the warehouse are queued here
Trucks from the queue are unloaded before leaving for the harbor

TABLE 6.3: Analysis of the Warehouse Model

complex XML types have been generated with the help of the framework, which are at the heart of every model pipeline simulation:

- *container-type:*

  Defines a container type and its generation cycle within the Harbor model.

- *container-creation:*

  Specifies a unique identification number for a container, as well as its type and time of creation.

- *truck-information:*

  Sets an identification number for a truck, its traveling speed and distance.

- *truck:*

  Holds a truck process identification, with the information for the container identification of the currently loaded cargo, and the respective arrival times at the harbor and warehouse.
- *container-handling:*

  Defines a handling duration for a specific container type.

- *container-storage:*

  Determines the time of storage for a container with a specific type and identification.

The resulting SOAP message used to communicate with the Dry Port WS is shown in tabular form in table 6.4. The interaction in between the Experiment WS, the composite WS and the atomic WSs are all handled by the data flowing through the pipeline like has been shown in chapter 3.

6.3.3 Model Pipeline Execution

A schematic representation of the approach for the Dry Port model pipeline example is shown in figure 6.8.

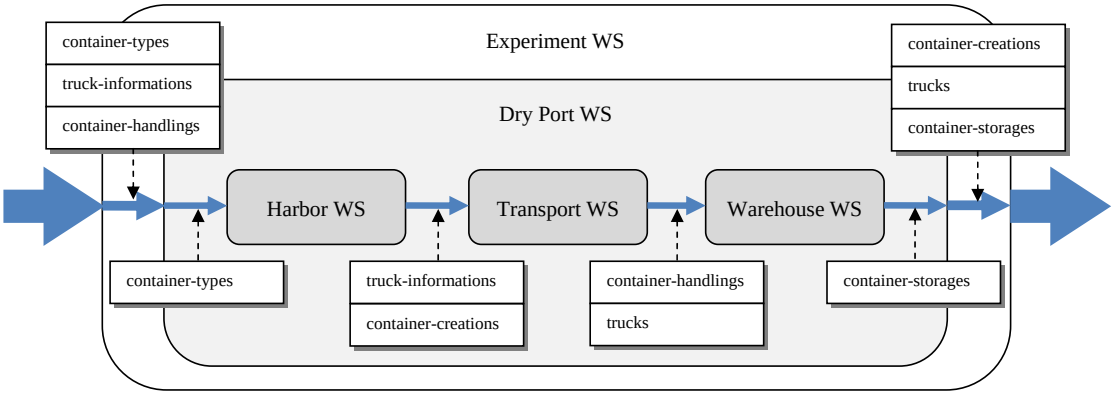


FIGURE 6.8: Data Flow in the Dry Port Model Pipeline

All the necessary input parameters are delivered to the Dry Port interface with an according request to the Experiment WS. The information about container types is sent to the Harbor WS, which generates the container creation times after the execution of

- model
  - name
  - description
  - runtime
  - parameters
    - parameter
      - container-types
        - container-type
          - type-designation
          - generation-interval
      - truck-informations
        - truck-information
          - truck-id
          - truck-speed
          - travel-distance
      - container-handlings
        - container-handling
          - handling-designation
          - handling-duration
  - outputs
    - output
      - container-creations
        - container-creation
          - container-id
          - container-designation
          - generation-timestamp
      - trucks
        - truck
          - truck-id
          - container-id
          - harbor-arrival
          - warehouse-arrival
      - container-storages
        - container-storage
          - storage-id
          - storage-designation
          - storage-timestamp

TABLE 6.4: SOAP Message for the Dry Port WS

its model. Together with the truck information, the Transport WS is then in turn able to execute its model to calculate the truck travel time stamps that record their processes. With the container handling data from the original input, the Warehouse WS can execute the model it interfaces with to compute the container storage times. All the produced output data is then delivered by the Dry Port WS to the Experiment WS, which creates an according SOAP response for the simulation user. This is the basic procedure for one run of the model pipeline simulation with no feedbacks.

It is however apparent that feedbacks between the models are occurring and are actually quite numerous. This is because in the model pipeline concept the models are not running concurrently and thus need to loop back information in the pipeline to communicate certain events to each other. Therefore the pessimistic feedback approach presented in chapter 5 has been applied to the Dry Port example, since it has already been established that it is more performant in those situations in comparison to the optimistic feedback mechanism, which should be used in low feedback situations.

In the Dry Port model pipeline, the feedback events are occurring between the Transport and Harbor WSs because of two restriction identified previously in the mode of operation of the Harbor model: no trucks can be loaded with cargo while the harbor is generating new containers, and no new containers can be generated while there is still cargo at the harbor. Therefore, the Harbor model must be informed equally about its own processes as well as those from the Transport model. Concretely, this means that the Harbor model must be aware of its past activities and the current position of the trucks to control its internal processes properly. To extend the model's knowledge base, the Harbor WS is reading the output data produced in previous runs to acquire the necessary additional information concerning previous container creation and truck activities, and their respective timestamps. The "generation-timestamp" of past container generations allows the Harbor model to cross-check with the current runtime if new containers can be produced, so that generating cargo remains impossible while old cargo has not been loaded onto a truck. Similarly, the "harbor-arrival" of the truck data type allows the Harbor model to detect if an unloaded truck has arrived at the harbor and entered the waiting queue.

As seen in chapter 4, the Experiment WS is able to propagate information about already computed output from the Harbor and Transport models back to the Harbor WS with the help of the IDS approach. It starts the simulation in a time frame of 0 to 1 time units, and then steadily increases that time frame with each run of the simulation, until it reaches the requirement originally set by the user in the "runtime" tag of the SOAP request sent to the Experiment WS. While this principle allows to recreate the interactions and general functionality of the Dry Port federation in model pipelines, repercussions concerning the execution times of the simulation are to be expected.

6.4 Simulation Runs and Load Tests

In this section the Dry Port simulation that now has an implementation available both with the HLA and model pipeline approach, are analyzed by studying their respective capabilities and general performance.

6.4.1 Data Comparison

Figures 6.9 and 6.10 are respective screen captures of the Dry Port federation and model pipeline after their execution.

These are obviously quite different. The HLA federation uses a viewer federate to display the processes of the simulation. It is has been adapted from the Sushi Restaurant example, and is basically just a passive recipient of the simulation data of the federation. The ring in the middle is a representation of the road the trucks navigate to go from the harbor to the warehouse and back. The harbor is displayed on the left hand side and the warehouse can be found on the right. Trucks also have a pictorial representation: while loaded trucks have a cargo icon displayed, empty trucks do not. During the federation execution, the viewer updates the display with each step using the newest information concerning the positioning of the trucks. The display for the model pipeline version of the Dry Port simulation is not quite as elaborate, since the XMPF has not the possibility to display real time data in a graphical animation due to the architecture's concept of

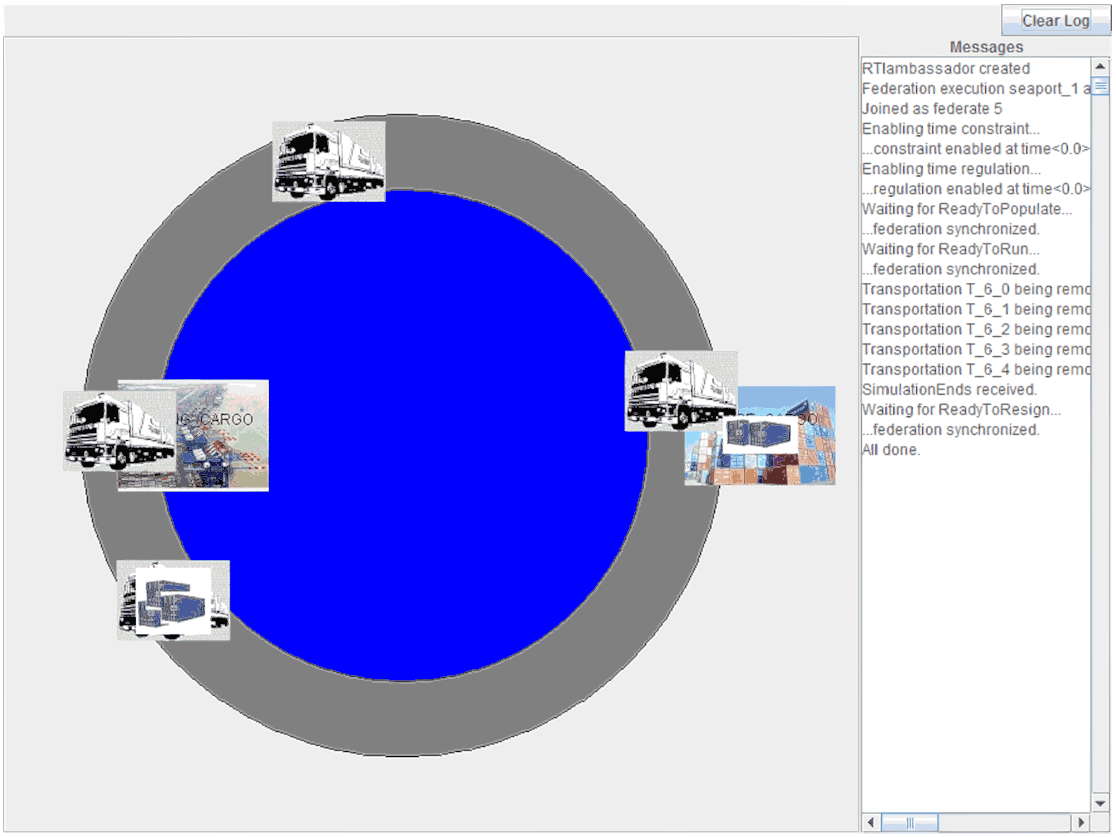


FIGURE 6.9: Viewer Federate of the Dry Port HLA Federation

having the models run sequentially. However, the SoapUI software used in chapter 4 is able to present the end results of a simulation run, which are enough to compare both coupling strategies. The screen capture shows the view the application offers after a successful run of the model pipeline. The comparison of that SOAP response with the log files of the Dry Port federation after it's execution show that the two simulations produce the same outputs, as demonstrated in table 6.5.

It shows the container generation times of the Harbor model for both coupling paradigms after the execution of a simulation run with the following parameters:

- A runtime of 600 time units.
- Three container types C1, C2 and C3 with a respective generation interval of 25, 30 and 50 time units and a respective handling duration of 20, 30 and 45 time units.
- Four trucks that drive at a speed of 60 distance units per time unit.

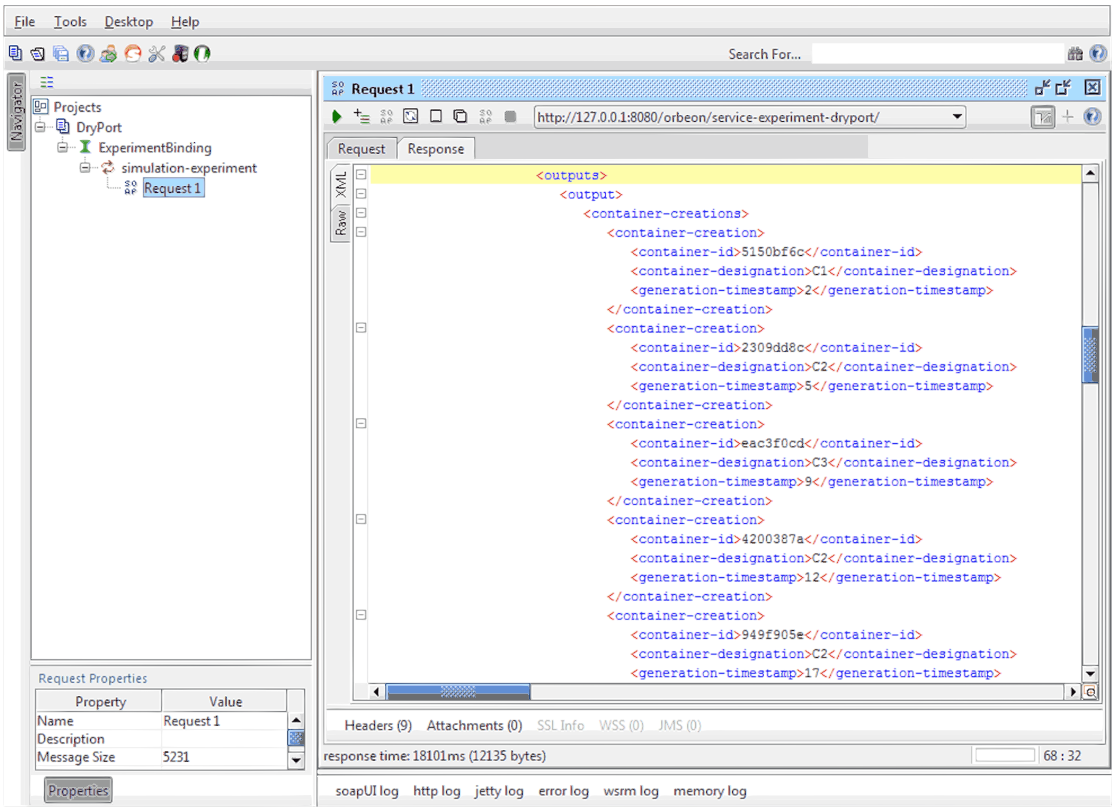


FIGURE 6.10: SOAP Response of the Dry Port Model Pipeline

- A road with a length of 500 distance units to get from the harbor to the warehouse and back.

The first column, generation-number, is just a way to identify each line and is not part of any output. The container-designation and generation-timestamp column however are directly taken from the model pipeline output, while the logfile-timestamp column is taken from the federation execution log files, as the name suggests. One can see that the generation timestamps for the containers from the model pipeline match those from the HLA federation. The HLA simulation uses doubles as data type to hold the timestamps. This was just an arbitrary choice made during the implementation, and it turned out that there was no added value to the information, which becomes apparent in the table. Therefore the decimal numbers present in the HLA output are not visible anymore in the model pipeline output, because of a change in data type from double to integer during the XML formatting specific to WSs. The matching of results can also

generation-number	container-designation	generation-timestamp	logfile-timestamp
1	C1	26	26.1
2	C1	52	52.1
3	C3	103	103.1
4	C1	129	129.1
5	C3	180	180.1
6	C3	231	231.1
7	C2	262	262.1
8	C3	313	313.1
9	C3	364	364.1
10	C1	390	390.1
11	C1	416	416.1
12	C2	447	447.1
13	C3	498	498.1
14	C3	549	549.1
15	C2	580	580.1

TABLE 6.5: Container Generation Times in the HLA and Model Pipeline Simulations

be observed for the Transport and Warehouse models but were omitted here, because they would offer very little additional information.

6.4.2 Performance Analysis

Now the relevant output dimensions have been compared and it has been ensured that both approaches deliver the same results, the next step is to compare the performance of the Dry Port model pipeline and federation. A total of five runs for each of the chosen runtimes of 25, 50, 100, 200 and 400 time units have been executed and measured for both approaches. That way a more conclusive interpretation of the performance

can be extracted from the mean wall-clock time. The collected readings are displayed in table 6.6 and 6.7. The measurements were made on a machine with the following specifications:

- Windows 7 Professional 32-bit
- Intel Core 2 Duo @ 1.83 GHz
- 4 GB DDR2 RAM

Runtime (tu)	**25**	**50**	**100**	**200**	**400**
Run 1 (ms)	3463	4368	5405	7124	12655
Run 2 (ms)	3433	4238	5687	7261	11142
Run 3 (ms)	3688	4076	5536	7714	11684
Run 4 (ms)	3356	4050	5668	7974	11524
Run 5 (ms)	3359	4033	5487	7792	11404
Mean value (ms)	3459.8	4153	5556.6	7573	11681.8
Variation coefficient (%)	0.04	0.04	0.02	0.05	0.05

TABLE 6.6: Execution Times for The Dry Port Federation

The measurements for the model pipeline and the HLA federation both show a linear trend for the execution times, which indicates a complexity of O(n). However, while the HLA federation execution time only grows by approximately 0.02 seconds per time unit increment, the model pipeline requires an additional 0.31 seconds. So in retrospect, both approaches do show a desirable complexity, but the HLA still has the advantage of a better performance. The coefficient of variation was calculated to verify that no

Runtime (tu)	25	50	100	200	400
Run 1 (ms)	6979	13628	26934	56119	123497
Run 2 (ms)	6585	13510	28228	56929	125525
Run 3 (ms)	6865	13582	26536	55950	118545
Run 4 (ms)	7206	13322	29025	57361	122884
Run 5 (ms)	6874	13504	27405	55894	128351
Mean value (ms)	6901.8	13509.2	27625.6	56450.6	123760.4
Variation coefficient (%)	0.03	0.01	0.04	0.02	0.03

TABLE 6.7: Execution Times for The Dry Port Model Pipeline

fluctuations were able to distort the different runs in any major fashion. Based on the measurements, the development of the mean execution time for the two architectures in the case of the Dry Port example is shown graphically in the figure 6.11.

6.4.3 Application field

In conclusion it can be stated that a transfer of an HLA federation to a model pipeline coupling is possible. But one has to accept a relatively significant decrease in performance because of the indirect communication approach in between the models of a pipeline. The questions arises for which context the model pipeline approach is suitable. Choosing the right approach for a distributed simulation depends on this context and the application environment. Due to the comparably poor performance measurements in contrast to the HLA, it seems reasonable to favor the HLA for the Dry Port subject. On the other hand, performance is not always the decisive issue, as other aspects can play a more important role. In the HLA, the simulation logic is strongly intertwined

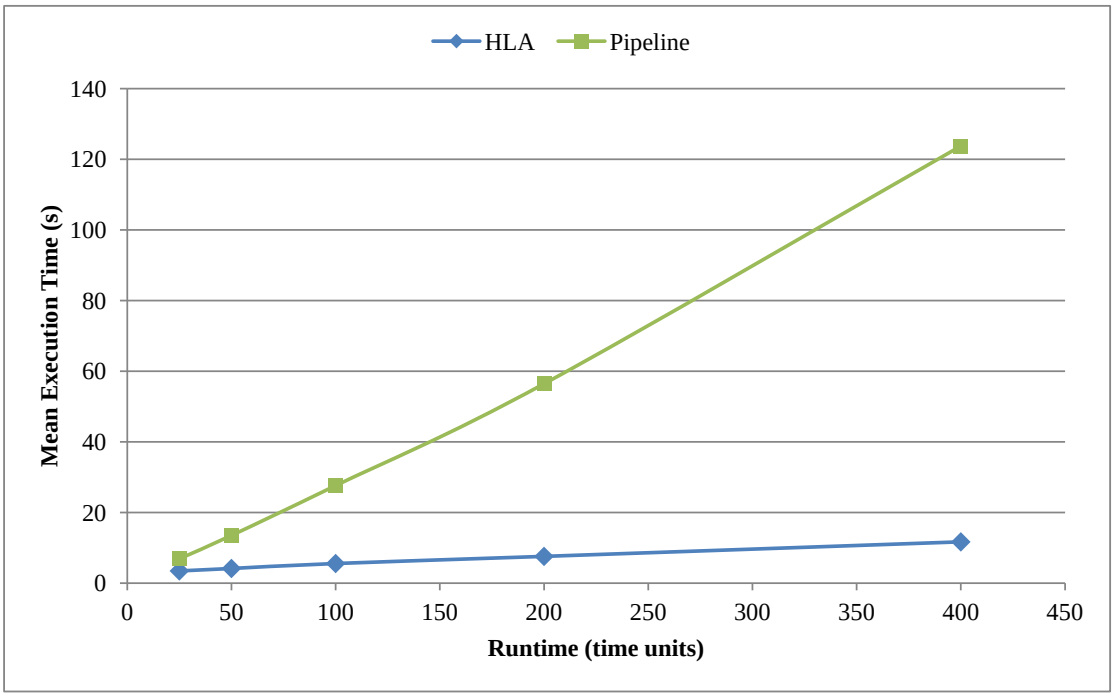


FIGURE 6.11: Graph of the Execution Times for The Dry Port Simulation

with the synchronization and messaging routines of the RTI. The addition of new federates to a federation is not a simple matter in practice. It requires a relatively extensive analysis and interpretation of the simulation inputs, outputs and processes. As has been shown in this chapter, the implementation of the Dry Port example as model pipeline was rather fast and simple with the already available models from the HLA federates. The resulting WS interfaces generated a lot less and simpler code compared to the Dry Port federation. While all in all, the model pipelines have created 2319 lines of code, of which most is automatically generated by the framework, the HLA federation requires 9884 lines of code. This fact is of interest mainly in practical applications due to possible time and cost constraints. There seems to be a trade-off between performance and ease of use, and the integration of the approach. That trade-off can change according to the application context and environment, and therefore may justify the choice of the approach. In the defense sector for example, where high performance simulations are used to simulate war scenarios, performance obviously play an essential role. In the case of commercial issues however, where short development cycles of products require to rely on minimal training and short implementation phases, easy handling and low integration problems have priority over fast execution times.

Chapter 7

Summary and Outlook

7.1 Summary

This section is a recapitulation of the subjects touched in this doctoral thesis. It encompasses the discussion of the first chapter concerning the current state-of-the-art, the demonstration of a working model pipeline prototype found in chapter two, the description of the synchronization concepts developed for model pipelines presented in chapter three, as well an explanation of the framework created to build model pipelines in chapter four, and the comparison of model pipelines with the HLA that took place in the previous chapter.

7.1.1 State-of-the-Art

Distributed simulation has been an important subject in the field of computer science since the late 70's [COS95]. Regarding the industrial and corporate world, it is represents an attractive approach to designing, managing and optimizing business and manufacturing processes [LAK00], because it promotes integrability, interoperability, scalability and reusability amongst other characteristics.

Distributed simulation plays a major role in the concept of distributed collaboration, due to the facilitated sharing of information and research. It allows to share expert

knowledge, which industrial and corporate organizations are heavily reliant on, across space and time [COM99]. Distributed collaborations therefore feature the advantages of an increased reach, responsiveness, adaptiveness and cost effectiveness. However, the concept also comes with some drawbacks. Information sharing in the corporate world, even more than in any other environment, is a touchy subject. Information security has to be ensured to protect an organization's assets. To that end, the LISI model defines five levels of interoperability that describe how an increased degree of information entanglement affects the way data is being shared [IWG98]. The model pipeline approach presented in this thesis adheres to the level 2 of interoperability according to that scale, to allow for a satisfactory level of interaction, while preserving the ability of the data holders to individually manage their own information. This falls under the category of interoperability on a conceptual level following the LCIM, which is based upon the LISI model and the NMI, which is a representation of the relation between the degrees of interoperability and its dimensions [TOM03]. This means that the dimension of composability is reached in the case of model pipelines, composability being the ability to combine modules in order to build semantically and syntactically correct simulation systems. To actualize interoperability however, one needs to understand the specifics and differences between the CSPs participating in a distributed simulation. The CSPI PDG has simplified that effort by releasing the current four IRM types, which allow on the one hand to identify the CSP interoperability capabilities of an existing distributed simulation, and on the other, specify the CSP interoperability requirements of a planned distributed simulation.

As already mentioned, distributed simulation allows for the reuse of software components, an idea introduced in the late 60's [MBN68]. It has gained a lot of acclaim in the corporate world, because it enables organizations to lower costs, reduce production time and increase quality. The most important factors in making components reusable are a high level of abstraction to separate implementation and interface of an application, and standards to allow components to interact. Two reuse strategies are currently used by software engineers: white-box and black-box reuse, the difference between the two being that white-box reuse allows developers to alter the code of a component. Black-box reuse does not expose any code and relies instead on interfaces, this eliminates the

need for complex documentations. Model pipelines therefore have incorporated that last approach for a simpler, streamlined development process.

Model pipelines are however not only an implementation, but an architecture. This thesis has therefore examined the most commonly used and well known structural patterns of architectures, the most relevant concepts being summarized in the following list:

- *Pipes and filters pattern:*
  A filter accepts a certain data input, transforms it, and produces an output that is transported over a pipe to the next filter.
- *Object-oriented pattern:*
  Components are made of instances of abstract objects that encapsulate data and methods. They communicate through explicit invocation of those methods.
- *Event-based pattern:*
  Modules can communicate by broadcasting events that implicitly invoke procedures. Those components are said to be in a publisher-subscriber relationship.
- *Layered pattern:*
  Hierarchical composition of layered systems interacting with each other through specific interfaces.
- *Component based pattern:*
  Encapsulated code artifacts communicating with each other to exchange data and provide an otherwise not available functionality.
- *Service-oriented pattern:*
  Units of program logic defined as services working autonomously, communicating trough specific set of common standards.

The model pipelines architecture, dubbed XMPF, is a heterogeneous composition of several of those patterns. First and foremost, it is based upon a component-based, service-oriented architecture. Simulation models are encapsulated in individual services that can be composed into bigger, more complex distributed simulations. The

services themselves are linked using the pipes and filter architecture, but internally exhibit the layered pattern, with the WS serving as a client for other WS in a pipeline, while providing a service to the underlying external CSP models.

In comparison, the HLA, which is an established architecture for the coupling of models in heterogeneous distributed simulations since its development for the DoD in 1996, uses a very different approach [DFW97]. It works as a layered system of object-oriented components interoperating through the implicit publish-and-subscribe mode of method invocation. A distributed simulation that uses the HLA is called a federation and consists of four components: the RTI, a SOM, FOM and external models know as federates. Both the federation and federates have to adhere to a strict set of federate and federation rules to be able to interact. However according to some sources, the bloated federate interface specification, can result in complex and error-prone implementations [STR06] [HHH12]. Model pipelines and the XMPF are a novel approach to realize distributed simulations in heterogeneous environments, which specifically aim to create a simple and reliable tool for collaborative efforts, like joint projects.

7.1.2 Model Pipeline Prototype

One such undertaking is the Airport2030 cluster of excellence project, commissioned by the FMER, which has served in this thesis as an example to show how a model pipeline operates. In the context of that project, new technological advances are being developed to improve upon the current ground handling procedures at the airport Hamburg. This currently represents a very pressing matter because of the projected increase of 300% in air traffic within the next 20 years, with minimal to no possibility of expansion of the local airport infrastructure [AIR11]. To this end, industry and research organizations have partnered up to generate models for the airport transit connection, terminal and apron with their unique expertise in the according fields. Ultimately, the goal is to combine those models into a simulation allowing the analysis of future advancements for the time horizons 2015 and 2030.

Each of the models of the simulation have been developed with different CSPs, each with varying modeling paradigms, detail and time resolutions. The TUHH developed a multi-modal traffic generation model for the transit connection, able to offer a representation the whole local traffic in and around the city of Hamburg. The results for a selected pre-cast of runs of that very complex model have been integrated as database into the model pipeline. The UHH has modeled the passenger and luggage flow in the two airport terminals of the Hamburg airport using MATLAB and the built-in SimEvent library. The apron model was modeled by the DLR, which calculates handling and travel times of vehicles on the runway and apron using MATLAB.

The model pipeline prototype has allowed to couple these models in a hierarchical fashion. The individual models are interfaced with WSs and grouped into a composed simulation for the airport as a whole using pipelines, which is also features a WS. That WS is controlled by the Experiment WS. It is at the root of every model pipeline simulation and serves as a common interface to conduct simulation runs with that new technology. It allows to sequentially execute and describe multiple runs, engage post-processing procedures and is responsible for performing the feedback processes in the model pipeline architecture.

A standard execution of the airport model chain with the model pipeline prototype works as follows (see figure 3.5 of chapter 3):

1. A SOAP request is sent to the Experiment WS.
2. The Experiment WS relays that message to the Airport WS.
3. The Airport WS in turn, relays the message to the WS of the first model in the pipeline, which is the transit connection model.
4. The transit connection model determines an appropriate scenario from the input, consisting of passenger data arriving at the airport terminals, to produce as output.
5. That output is delivered back to the airport WS, which contacts the next WS in the pipeline.

6. The terminal WS receives the output produced by the transit connection model and creates its own results based upon them, consisting of passenger and luggage time stamps generated during their travels inside the terminals.

7. Again, that output is delivered back to the airport WS, which contacts the apron WS, which is the final WS in the pipeline.

8. Based upon the passenger and luggage time stamps from the terminal model, the apron model calculates the vehicle movements on the apron.

9. That information is sent back to the airport WS.

10. Finally, the airport WS sends the results of the whole chain back to the Experiment WS.

11. The Experiment WS is now able to build a SOAP response to the original request from the model pipeline user.

To facilitate the operation of the model pipeline prototype for the user, the XMPF framework provides the ability to generate custom web-GUIs matching specific pipelines. Such a web-GUI was implemented for the Airport2030 project so that all the participating partners could access the simulation from everywhere over the Internet, over a standardized form. Adding that functionality to the XMPF was relatively easy due to the fact that WSs, which are a central part of model pipelines, produce and consume XML. Input and output representation as forms can be derived from that data using XForms to generate XHTML web pages and SVG graphics.

7.1.3 Feedback Mechanisms

Although the information flow in the model pipeline prototype for the Airport2030 project is generally forward-oriented, the technology itself also offers the possibility for models of a distributed simulation to interact with each other, even if they are not ordered chronologically in the chain they build, thanks to the implemented feedback mechanisms.

To establish some basics, this thesis has summarized the different notions of times used in simulation: physical, simulation and wall-clock time. Afterwards, the execution modes of simulations have been listed, which are scaled real-time, real-time and as-fast-as-possible. Then, the most common time-flow mechanisms for discrete simulations have been touched upon: time-stepped and event-driven executions. Similarly, receive-order and timestamp-order delivery have been described, being two of the most widely applied delivery mechanisms. Finally, synchronization techniques such as conservative, optimistic, relaxed and combined synchronization have been discussed.

More importantly, the two dominant forms of communication in distributed simulation have been presented, to establish the fundamentals to compare classical coupling mechanisms to the approach used by model pipelines. It has been established that message passing can in fact be implemented using shared memory and is at its core very similar to communication with shared variables. In that case, models of a distributed simulation exchange information using a global data area accessible to all participants. The drawback with that concept is that since the models all run concurrently, the runtime system has to synchronize all their local wall-clock times. Model pipelines can avoid that additional effort by running the models they couple sequentially, resulting in an overall less complex architecture. This simplification however comes at a price. Models generating information of importance to other models situated before in the pipeline, cannot make that data available to them without feedback capabilities.

The feedback approaches implemented in model pipelines are based upon the idea of rolling back a simulation to a state were all variables hold consistent values, called state-saving. Since creating and managing such save-states is rather complicated [PER06], and model pipelines aim to build a simple and straightforward architecture, the implemented approach limits restore points to the start of a simulation [HIW10]. The concept comes in two versions that tackle the feedback problem slightly differently: IDS and OIDS. The IDS, as seen in table 4.1 of chapter 4, uses multiple runs that incrementally advance the simulation time. With each consecutive run, the output of all previous runs is added as input to the new run, so that the interfaces of the individual models of the overall simulation can adapt the parameters for their sequential execution. Due to the reiteration of previous runs during a simulation however, the advantage of foregoing

restore points comes at the cost of processing time and bandwidth. This effect can be toned down for simulations with an expectedly low amount of feedback events with OIDS. This approach also uses reruns of a pipeline to handle feedback requests, but only initiates these reruns under certain circumstances. The Experiment WS as an administrative component, is empowered to detect whether a feedback loop is in order or not. To do so, it scans the final output of each run for feedback requests in the produced XML file. Should a WS interface have created such a request, and it was not handled before the end of the run, the Experiment WS re-executes the model pipeline fully, so that the addressee may process that request. The downside is here, is that simulations with a high occurrence of feedbacks will run even slower than with IDS, since each run will be executed fully, because the optimistic approach of the OIDS does not expect such a behavior.

The measurements done in chapter 3 confirm that assumption. The model pipeline prototype created for the Airport2030 project presented in chapter 2 was used to compare the two different implementations of the idea to use reruns as save-states for simulations. The wall-clock time was measured using both approaches for the same simulation. It turned out that the OIDS is performing better for simulations where the following equation applies:

$$x < \frac{n-1}{2} \tag{7.1}$$

x is the number of runs necessary to process all feedback events, while n is the maximum number of runs at a selected step size. That means that with 12 or more reruns to handle feedback requests with OIDS, the simulation would execute faster with IDS.

7.1.4 Model Pipeline Framework

The XMPF has been built using a multitude of widespread and established technologies, integrated in the Orbeon Forms platform [ORB11]. Since it implements a mature XML

pipeline engine introducing XPL, which is a W3C recommendation, it has proven to be a good choice to build the model pipeline architecture, instead of developing a proprietary solution. Orbeon Forms is using XHTML and XForms for presentation purposes, XPL, XSLT and JAVA for the program logic, and XML, XPath and XQuery for the data management. This is reflected by the fact that the platform is built around the MVC design pattern, which dictates that each application is divided into three categories of components: models, views and controllers.

The XMPF is used to build WS interfaces for external models to allow coupling them over a client-server architecture into a composed simulation. Those WS all feature a page flow controller, an XML schema, a WS description and an XPL pipeline. Some WS also employ JAVA processors to build the connection to their model. All these files have a very standardized structure that can allows them to be built automatically with relatively little user input. How they interact can be seen in figure 5.8 of chapter 5. Three types of WS can be distinguished:

1. *Experiment WS:* Unique WS with managing mechanisms for runtime, feedbacks and multiple runs of composite WSs
2. *Atomic WS:* WS interfacing directly with external CSPs
3. *Composite WS:* WS interfaces to compositions of other WSs, to form a chain of models

How these WSs work together can be seen in figure 5.9 of chapter 5. Furthermore, the XMPF can also be used to build web-GUIs for model pipelines. These also come with a fixed number files with a standardized structure. A page flow controller, a form view, a result view and an XPL pipeline. Their interaction allows, shown in figure 5.15 of chapter 5, allows to send input data to a model pipeline though a web form, and display the output that comes back from it in a easy to read, tabular fashion.

As already mentioned, the XMPF is able to generate all those files with minimal user input. All that is required, is the specification of base, simple and complex data types, feedback requests and responses, and import sources that the model pipeline being built should be able to process. Based on that information, the framework can create the files

that make up a model pipeline and a matching web-GUI. To facilitate the task for the user even further, a graphical development tool for the XMPF has been implemented. It allows to define all the necessary parameters for a new model pipeline from an accessible form. The application furthermore allows to save and load projects, so that existing pipelines may be edited more easily when necessary. Further improvements of the current prototype are envisaged, to give the user the ability to also record manual changes to the code of the files that constitute the model pipeline.

7.1.5 HLA Versus Model Pipelines

The main problem with the conversion of an HLA federation to a model pipeline arises from the conceptual differences between the two distributed simulation approaches. An HLA federation uses a message based communication to allow the distinct federates to interact. All those messages are managed by a central unit known as the RTI. The federates merely register as publisher and subscriber for specific information. They are then notified about the actions of other models by receiving updates of that information through the RTI. In the model pipeline approach, such a central managing instance is not available. The models are instead executed in a stringent sequence. The output of each model is then delivered as data to feed the input of one of the next models in that sequence. Regardless, it has been shown that interactions from the HLA can be mapped to model pipelines thanks to their feedback mechanisms. The only real difference here, are that they communicate directly, instead of using a central managing instance. This had only a marginal effect on the distributed simulation conversion, since only a smaller portion of logic had to be added to each model WS interface to work with the altered situation. To underline the differences between the two concepts, a description of a standard scenario for both will be presented in the following:

- *HLA:*

  The warehouse determines via an internal method that a truck has arrived. If the warehouse is not busy, it will unload the truck and take responsibility of the cargo. The warehouse publishes the action to the RTI and starts the storing process. The RTI distributes a message to all federates that have subscribed to the cargo object type. The Transport federate receives that message and changes the state of the truck to "empty", so that it may leave the queue at the warehouse and drive to the harbor.

- *Model Pipelines:*

  The warehouse is able to detect all the trucks through the XML input. At a specific point during the execution, the runtime and arrival time at the warehouse for a truck matches up. If the warehouse is not busy, a storing event is created and written to the output. At first, the Transport model is unaware of this. But on the next iteration of the model pipeline execution, the Harbor model is able to extract from the XML output if a storing event has happened by analyzing the cargo IDs. If an event has been found with a cargo ID matching the one transported by a truck, it moves that truck from the warehouse to the harbor.

As one can see with the help this scenario, the processes in the cases of the HLA and model pipelines are rather similar. First, the warehouse determines if a truck has arrived. If the warehouse is not occupied it will engage the storing process. In both approaches, the Transport model is not aware of this at first. In the HLA it is informed thanks to the publisher-subscriber concept. With model pipelines, the Transport model is able to read actions from the warehouse from the output it produces. Afterwards both approaches work identically again: the Transport model removes the truck from the queue at the warehouse and moves it to the harbor.

The previous example also suggests that model pipelines have a worse performance than the HLA. A model pipeline has to run several iterations of a simulation for the feedback mechanisms to work. These are necessary so that the indirect messages located in the output can be read by the different models. The HLA however, only creates

and processes interactions when necessary and at the actual moment a change in the system state happens. On the other hand, because model pipelines renounce such a design choice, they circumvent the need for global time stepping and synchronization. The load tests run in the last chapter managed to confirm that the presumed loss of performance with model pipelines is a reality. In contrast to the HLA, the execution times for the Dry Port simulation increased rapidly with a prolonged runtime. As already noted, this can be explained by the additional effort for the processing of previous outputs and the identification of relevant information.

However, one must reflect upon the operational area of a distributed simulation. In fields where as-fast-as-possible executions are required, the HLA clearly represents the better option. On the other hand, if short development times and ease of use are a priority, model pipelines would seem to be the wiser choice. Indeed, although more difficult to quantify, experience through research has shown that their implementation is less time consuming and complex than an HLA approach. Similarly, their operation is objectively more intuitive because the option to automatically create standardized web-UI structures.

It is a well-known fact that simulation does not yet find enough appreciation in the industry. And when it is employed, it is generally implemented with a monolithic approach [STR06]. Considering the increasing globalization and networking in between corporations, distributed simulation concepts will be on demand sooner rather than later. The idea of model pipelines might open new ways to weave this IT specific discipline into industry standards. The fear of loss of know-how could be mitigated by the encapsulation of models through WSs and the general black-box nature of the architecture. All the actual knowledge is integrated in the models and can only be accessed through predefined interfaces. Any specifics about the implementation cannot be seen from the outside, just called upon. Thusly, cross-company collaborations become possible without having to share more information than required. Especially in relation to major projects, an enormous potential can be identified to optimize company processes and procedures. Furthermore, the relatively easy integration and connectivity of the model pipeline services over Internet protocols, makes this technology easily applicable in virtually any situation. Because of the use of the many, well defined WS standards, there

is no need for dedicated uniform object models, common language or proprietary platforms. Hence, the integration of model pipelines is much easier than, say, supplying the RTI for several, possibly differing systems. And because of the use of open standards, it makes the model pipeline approach also more cost-effective.

7.2 Outlook

Finally, this last section is used to discuss the possible future developments of model pipelines and the XMPF more specifically. This covers modular extensions to the framework, improvements to the automated creation of JAVA processors, ideas to increase the performance and security of the pipelines, and enhancements to the XMPF IDE.

7.2.1 Modules

Modularity is an important factor in many areas, including technological and organizational systems. Since the 60's people of the software engineering world have been using libraries and frameworks to concentrate on the actual logic of their applications, to shorten the cost and time needed for development [BAC00], [JAC00]. Another advantage is the improvement of the adaptability of the created software by decoupling design artifacts, thereby hiding their information [PAR72].

Against this backdrop, the XMPF is prepared to accommodate modules to facilitate a WS's communication with external simulators, by providing standardized interfaces for given CSP's. The familiar issue of CSP's having differing programming paradigms also comes with the problem of offering varying modes of interaction with other applications, if any at all. To this end, it is often necessary for the XMPF to provide an interface to the CSP a specific atomic WS is to interact with.

An example for one such module has been introduced briefly in chapter two. The JaMaLa application is a JAVA-MATLAB server that implements the undocumented JAVA-MATLAB Interface (JMI) included with MATLAB [KAP10]. It is a standalone

application that acts as an interface to MATLAB, which is able to be controlled using appropriate commands sent via the TCP/IP protocol [FWH12]. Figure 7.1 shows a basic representation of the inner workings of JaMaLa.

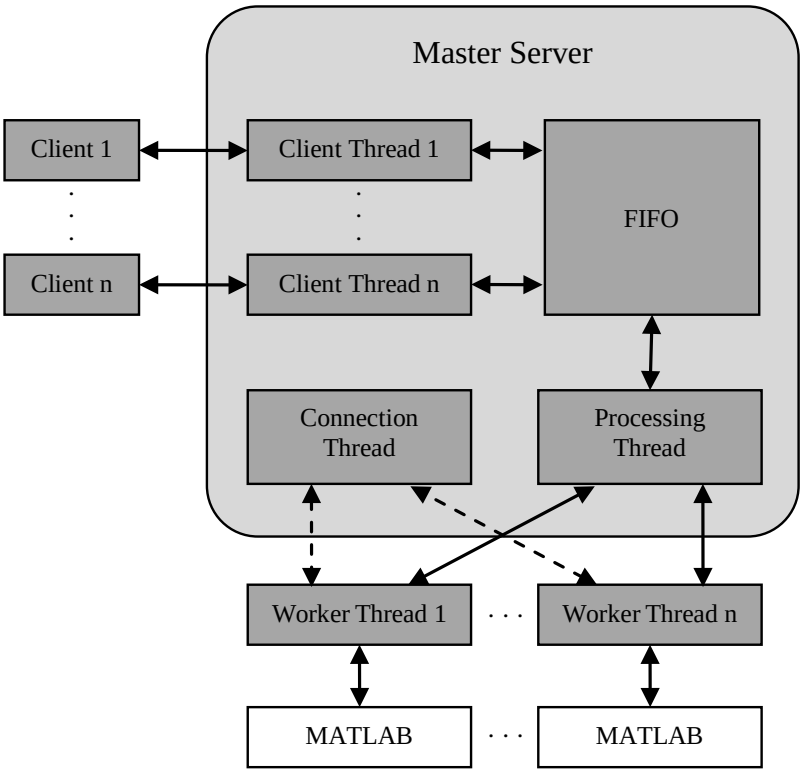


FIGURE 7.1: Functionality of JaMaLa (confer [FWH12])

A master server is responsible for the communication with clients, as well as the distribution and forwarding of requests to worker servers, using a First-In-First-Out (FIFO) approach. The worker servers execute the simulation models on a MATALAB session they call and manage. After a simulation has been executed, its output is returned to the master server, which forwards it to the respective client.

The Java processors that can operate in XMPF pipelines, have currently the option to call upon the necessary methods to interact with the JaMaLa master server and thus execute MATLAB models. The sum of these methods, the JaMaLa master server and worker servers constitute a module that was developed to facilitate the use of Mathworks MATLAB with the XMPF. Other CSPs are not supported by modules at this point in time, but the library can be extended ad infinitum with interfaces that support communication over files, messages or even shared memory.

It is also possible to add post-processing functionalities to model pipelines through modules offered as WS. In the context of the Airport 2030 project presented in chapter 2 for example, the UHH is currently developing a Soft Computing Framework (SCF) that will be able to solve optimization problems through approximation [ZAD94]. It is designed to be called as WS to complement any model pipeline simulation by further processing their output.

7.2.2 Usability

There is an implementational concept that was adopted for the XMPF in its current state that would benefit from an overhaul. As it stands, the XML information received and created by Java processors is stored in multi-dimensional lists. This is fine for small entities, but the automatically generated code can quickly become convoluted and unintelligible with the growing complexity of simulation data sets, which makes creating and managing the Java processors more difficult than it ought to be.

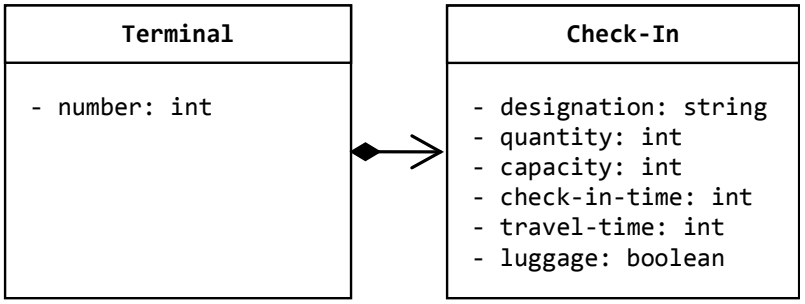


FIGURE 7.2: UML Diagram of the Terminal Class Containing the Check-In Class

In an effort to improve upon some the main aspects of the XMPF, which are simplicity and usability, it is planned to update the approach described above by switching from lists to full-fledged objects. Each data set from the input XML and for the output XML shall be translated into a matching object class with attributes, which may have a compositional inter-class relationship with other classes. Taking as an example the input parameters for the Airport WS shown in table 3.1 of chapter table 3, a terminal entity as made up of the terminal-number attribute and check-ins. The idea is therefore for the future, to let the XMPF create automatically the two classes seen in the UML diagram

in figure 7.2. It is expected that this should speed up and ease development of custom Java processors for model pipelines, as well as facilitate their maintenance.

7.2.3 Performance

With growing complexity, a model might eventually need more input, and produce accordingly more data. With the information transfer in the XMPF being based upon XML data, transmission times may increase the overall execution time of a model pipeline simulation. An approach currently being explored to lessen that effect, is to employ HTTP compression mechanisms.

This mechanisms make it possible for web servers and clients to optimize their bandwidth usage to provide greater transmission speeds. The HTTP data is compressed by the server, the most common principles used currently being Gzip [FSF13] and deflate [DEU96]. A research report on HTTP compression shows that the average website can achieve 27 percent in bytes reductions [MCL02]. It must be investigated how much this reductions can shorten model pipeline execution times.

7.2.4 Security

Security is a concern when using WSs, as the XMPF does. It does not provide yet protection against security risks like eavesdropping, message tempering or denial-of-service attacks. There are currently several WS security standards available, including XML Digital Signatures and Encryption, Security Assertion Markup Language, and WS-Security [PEL03]. They all try to implement message level integrity and confidentiality for WSs. WS-Security for example, is a family of different specifications, that allow to identify the origin of a message securely, detect that no one has tampered with the message data and that only the intended recipients of a message are able to read them. More information can be found in [WEE05] and [OAS02]. The XMPF is to be equipped with the according security measures in the future.

7.2.5 Editor

It would furthermore be interesting to expand upon the XMPF development GUI. As it stands, the interface is able to visualize the code created by the framework for the different files needed by model pipelines in a tree-like representation. The next step to take is to facilitate the direct input of user generated code into the automatically created one through a text-editor, effectively allowing code alterations like in an Integrated Development Environment (IDE). This would obviously simplify the design and management of model pipeline simulation projects.

Appendix A

Example SOAP Request

```
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <experiment>
      <date>
        2013-10-20
      </date>
      <description>
        SOAP Request Example
      </description>
      <adress>
        http://134.100.10.53:8080/orbeon/service-airport2/
      </adress>
      <iteration>
        pessimistic
      </iteration>
      <model xsi:type="sch:Model"
             xmlns:sch="http://134.100.10.53:8080/orbeon/service-airport2/schema">
        <name xsi:type="sch:NameString">
          Airport2030
        </name>
        <description xsi:type="xsd:string">
          Appendix A
        </description>
        <runtime xsi:type="xsd:int">
          24
        </runtime>
        <parameters>
          <parameter>
            <scenario xsi:type="sch1:Scenario">
              <name xsi:type="xsd:string">
                1
              </name>
              <description xsi:type="xsd:string">
                Test
              </description>
```

```
<weekday xsi:type="sch1:DayString">
  Wednesday
</weekday>
<ticket-price xsi:type="sch1:ScenarioInt">
  1
</ticket-price>
<areal-connection xsi:type="sch1:ScenarioInt">
  1
</areal-connection>
<substitution-effect xsi:type="sch1:ScenarioInt">
  1
</substitution-effect>
<demographic-change xsi:type="sch1:ScenarioInt">
  1
</demographic-change>
<business-traveller-ratio xsi:type="sch1:ScenarioInt">
  1
</business-traveller-ratio>
<enterprise-legislature xsi:type="sch1:ScenarioInt">
  1
</enterprise-legislature>
<extension-capabilities xsi:type="sch1:ScenarioInt">
  1
</extension-capabilities>
<economical-evolution xsi:type="sch1:ScenarioInt">
  1
</economical-evolution>
<class-division xsi:type="sch1:ScenarioInt">
  1
</class-division>
<security-legislature xsi:type="sch1:ScenarioInt">
  1
</security-legislature>
<emission-legislature xsi:type="sch1:ScenarioInt">
  1
</emission-legislature>
<plane-size xsi:type="sch1:ScenarioInt">
  1
</plane-size>
<airport-alliances xsi:type="sch1:ScenarioInt">
  1
</airport-alliances>
<non-aviation-revenues xsi:type="sch1:ScenarioInt">
  1
</non-aviation-revenues>
<arrival-departure-technologies xsi:type="sch1:ScenarioInt">
  1
</arrival-departure-technologies>
<apron-technologies xsi:type="sch1:ScenarioInt">
  1
</apron-technologies>
<passenger-handling-technologies xsi:type="sch1:ScenarioInt">
  1
</passenger-handling-technologies>
<plane-handling-technologies xsi:type="sch1:ScenarioInt">
  1
</plane-handling-technologies>
<energy-propulsion-technologies xsi:type="sch1:ScenarioInt">
  1
</energy-propulsion-technologies>
```

```
        </scenario>
      </parameter>
      <parameter>
        <terminals>
          <terminal xsi:type="sch:Terminal">
            <terminal-number xsi:type="xsd:int">
              1
            </terminal-number>
            <check-ins>
              <check-in>
                <designation xsi:type="sch:DesignationString">
                  counter
                </designation>
                <check-in-quantity xsi:type="xsd:int">
                  18
                </check-in-quantity>
                <check-in-capacity xsi:type="xsd:int">
                  40
                </check-in-capacity>
                <check-in-time xsi:type="xsd:int">
                  3
                </check-in-time>
                <check-in-luggage xsi:type="xsd:int">
                  0
                </check-in-luggage>
                <check-in-travel-time xsi:type="xsd:int">
                  3
                </check-in-travel-time>
              </check-in>
              <check-in>
                <designation xsi:type="sch:DesignationString">
                  automatic
                </designation>
                <check-in-quantity xsi:type="xsd:int">
                  10
                </check-in-quantity>
                <check-in-capacity xsi:type="xsd:int">
                  30
                </check-in-capacity>
                <check-in-time xsi:type="xsd:int">
                  3
                </check-in-time>
                <check-in-luggage xsi:type="xsd:int">
                  0
                </check-in-luggage>
                <check-in-travel-time xsi:type="xsd:int">
                  3
                </check-in-travel-time>
              </check-in>
              <check-in>
                <designation xsi:type="sch:DesignationString">
                  online
                </designation>
                <check-in-quantity xsi:type="xsd:int">
                  10000
                </check-in-quantity>
                <check-in-capacity xsi:type="xsd:int">
                  10000
                </check-in-capacity>
```

```
      <check-in-time xsi:type="xsd:int">
        0
      </check-in-time>
      <check-in-luggage xsi:type="xsd:int">
        0
      </check-in-luggage>
      <check-in-travel-time xsi:type="xsd:int">
        0
      </check-in-travel-time>
    </check-in>
    <check-in>
      <designation xsi:type="sch:DesignationString">
        night-before
      </designation>
      <check-in-quantity xsi:type="xsd:int">
        10000
      </check-in-quantity>
      <check-in-capacity xsi:type="xsd:int">
        10000
      </check-in-capacity>
      <check-in-time xsi:type="xsd:int">
        0
      </check-in-time>
      <check-in-luggage xsi:type="xsd:int">
        0
      </check-in-luggage>
      <check-in-travel-time xsi:type="xsd:int">
        0
      </check-in-travel-time>
    </check-in>
    <check-in>
      <designation xsi:type="sch:DesignationString">
        counter
      </designation>
      <check-in-quantity xsi:type="xsd:int">
        60
      </check-in-quantity>
      <check-in-capacity xsi:type="xsd:int">
        45
      </check-in-capacity>
      <check-in-time xsi:type="xsd:int">
        5
      </check-in-time>
      <check-in-luggage xsi:type="xsd:int">
        1
      </check-in-luggage>
      <check-in-travel-time xsi:type="xsd:int">
        3
      </check-in-travel-time>
    </check-in>
    <check-in>
      <designation xsi:type="sch:DesignationString">
        automatic
      </designation>
      <check-in-quantity xsi:type="xsd:int">
        30
      </check-in-quantity>
      <check-in-capacity xsi:type="xsd:int">
        45
      </check-in-capacity>
```

```
      <check-in-time xsi:type="xsd:int">
        1
      </check-in-time>
      <check-in-luggage xsi:type="xsd:int">
        1
      </check-in-luggage>
      <check-in-travel-time xsi:type="xsd:int">
        3
      </check-in-travel-time>
    </check-in>
    <check-in>
      <designation xsi:type="sch:DesignationString">
        online
      </designation>
      <check-in-quantity xsi:type="xsd:int">
        10000
      </check-in-quantity>
      <check-in-capacity xsi:type="xsd:int">
        10000
      </check-in-capacity>
      <check-in-time xsi:type="xsd:int">
        0
      </check-in-time>
      <check-in-luggage xsi:type="xsd:int">
        1
      </check-in-luggage>
      <check-in-travel-time xsi:type="xsd:int">
        3
      </check-in-travel-time>
    </check-in>
    <check-in>
      <designation xsi:type="sch:DesignationString">
        night-before
      </designation>
      <check-in-quantity xsi:type="xsd:int">
        10000
      </check-in-quantity>
      <check-in-capacity xsi:type="xsd:int">
        10000
      </check-in-capacity>
      <check-in-time xsi:type="xsd:int">
        0
      </check-in-time>
      <check-in-luggage xsi:type="xsd:int">
        1
      </check-in-luggage>
      <check-in-travel-time xsi:type="xsd:int">
        0
      </check-in-travel-time>
    </check-in>
    <check-in>
      <designation xsi:type="sch:DesignationString">
        luggage
      </designation>
      <check-in-quantity xsi:type="xsd:int">
        35
      </check-in-quantity>
      <check-in-capacity xsi:type="xsd:int">
        45
      </check-in-capacity>
```

```
            <check-in-time xsi:type="xsd:int">
              2
            </check-in-time>
            <check-in-luggage xsi:type="xsd:int">
              1
            </check-in-luggage>
            <check-in-travel-time xsi:type="xsd:int">
              1
            </check-in-travel-time>
          </check-in>
        </check-ins>
      </terminal>
      <terminal xsi:type="sch:Terminal">
        <terminal-number xsi:type="xsd:int">
          2
        </terminal-number>
        <check-ins>
          <check-in>
            <designation xsi:type="sch:DesignationString">
              counter
            </designation>
            <check-in-quantity xsi:type="xsd:int">
              18
            </check-in-quantity>
            <check-in-capacity xsi:type="xsd:int">
              40
            </check-in-capacity>
            <check-in-time xsi:type="xsd:int">
              3
            </check-in-time>
            <check-in-luggage xsi:type="xsd:int">
              0
            </check-in-luggage>
            <check-in-travel-time xsi:type="xsd:int">
              3
            </check-in-travel-time>
          </check-in>
          <check-in>
            <designation xsi:type="sch:DesignationString">
              automatic
            </designation>
            <check-in-quantity xsi:type="xsd:int">
              10
            </check-in-quantity>
            <check-in-capacity xsi:type="xsd:int">
              30
            </check-in-capacity>
            <check-in-time xsi:type="xsd:int">
              3
            </check-in-time>
            <check-in-luggage xsi:type="xsd:int">
              0
            </check-in-luggage>
            <check-in-travel-time xsi:type="xsd:int">
              3
            </check-in-travel-time>
          </check-in>
```

```
<check-in>
  <designation xsi:type="sch:DesignationString">
    online
  </designation>
  <check-in-quantity xsi:type="xsd:int">
    10000
  </check-in-quantity>
  <check-in-capacity xsi:type="xsd:int">
    10000
  </check-in-capacity>
  <check-in-time xsi:type="xsd:int">
    0
  </check-in-time>
  <check-in-luggage xsi:type="xsd:int">
    0
  </check-in-luggage>
  <check-in-travel-time xsi:type="xsd:int">
    0
  </check-in-travel-time>
</check-in>
<check-in>
  <designation xsi:type="sch:DesignationString">
    night-before
  </designation>
  <check-in-quantity xsi:type="xsd:int">
    10000
  </check-in-quantity>
  <check-in-capacity xsi:type="xsd:int">
    10000
  </check-in-capacity>
  <check-in-time xsi:type="xsd:int">
    0
  </check-in-time>
  <check-in-luggage xsi:type="xsd:int">
    0
  </check-in-luggage>
  <check-in-travel-time xsi:type="xsd:int">
    0
  </check-in-travel-time>
</check-in>
<check-in>
  <designation xsi:type="sch:DesignationString">
    counter
  </designation>
  <check-in-quantity xsi:type="xsd:int">
    60
  </check-in-quantity>
  <check-in-capacity xsi:type="xsd:int">
    45
  </check-in-capacity>
  <check-in-time xsi:type="xsd:int">
    5
  </check-in-time>
  <check-in-luggage xsi:type="xsd:int">
    1
  </check-in-luggage>
  <check-in-travel-time xsi:type="xsd:int">
    3
  </check-in-travel-time>
</check-in>
```

```
<check-in>
  <designation xsi:type="sch:DesignationString">
    automatic
  </designation>
  <check-in-quantity xsi:type="xsd:int">
    30
  </check-in-quantity>
  <check-in-capacity xsi:type="xsd:int">
    45
  </check-in-capacity>
  <check-in-time xsi:type="xsd:int">
    1
  </check-in-time>
  <check-in-luggage xsi:type="xsd:int">
    1
  </check-in-luggage>
  <check-in-travel-time xsi:type="xsd:int">
    3
  </check-in-travel-time>
</check-in>
<check-in>
  <designation xsi:type="sch:DesignationString">
    online
  </designation>
  <check-in-quantity xsi:type="xsd:int">
    10000
  </check-in-quantity>
  <check-in-capacity xsi:type="xsd:int">
    10000
  </check-in-capacity>
  <check-in-time xsi:type="xsd:int">
    0
  </check-in-time>
  <check-in-luggage xsi:type="xsd:int">
    1
  </check-in-luggage>
  <check-in-travel-time xsi:type="xsd:int">
    3
  </check-in-travel-time>
</check-in>
<check-in>
  <designation xsi:type="sch:DesignationString">
    night-before
  </designation>
  <check-in-quantity xsi:type="xsd:int">
    10000
  </check-in-quantity>
  <check-in-capacity xsi:type="xsd:int">
    10000
  </check-in-capacity>
  <check-in-time xsi:type="xsd:int">
    0
  </check-in-time>
  <check-in-luggage xsi:type="xsd:int">
    1
  </check-in-luggage>
  <check-in-travel-time xsi:type="xsd:int">
    0
  </check-in-travel-time>
</check-in>
```

```
            <check-in>
              <designation xsi:type="sch:DesignationString">
                luggage
              </designation>
              <check-in-quantity xsi:type="xsd:int">
                35
              </check-in-quantity>
              <check-in-capacity xsi:type="xsd:int">
                45
              </check-in-capacity>
              <check-in-time xsi:type="xsd:int">
                2
              </check-in-time>
              <check-in-luggage xsi:type="xsd:int">
                1
              </check-in-luggage>
              <check-in-travel-time xsi:type="xsd:int">
                1
              </check-in-travel-time>
            </check-in>
          </check-ins>
        </terminal>
      </terminals>
    </parameter>
    <parameter>
      <gates>
        <gate xsi:type="sch:Gate">
          <gate-number xsi:type="xsd:int">
            1
          </gate-number>
          <passenger-gate-travel-time xsi:type="xsd:int">
            1
          </passenger-gate-travel-time>
          <luggage-gate-travel-time xsi:type="xsd:int">
            1
          </luggage-gate-travel-time>
        </gate>
        <gate xsi:type="sch:Gate">
          <gate-number xsi:type="xsd:int">
            2
          </gate-number>
          <passenger-gate-travel-time xsi:type="xsd:int">
            1
          </passenger-gate-travel-time>
          <luggage-gate-travel-time xsi:type="xsd:int">
            1
          </luggage-gate-travel-time>
        </gate>
        <gate xsi:type="sch:Gate">
          <gate-number xsi:type="xsd:int">
            3
          </gate-number>
          <passenger-gate-travel-time xsi:type="xsd:int">
            1
          </passenger-gate-travel-time>
          <luggage-gate-travel-time xsi:type="xsd:int">
            1
          </luggage-gate-travel-time>
        </gate>
```

```
            <gate xsi:type="sch:Gate">
              <gate-number xsi:type="xsd:int">
                4
              </gate-number>
              <passenger-gate-travel-time xsi:type="xsd:int">
                1
              </passenger-gate-travel-time>
              <luggage-gate-travel-time xsi:type="xsd:int">
                1
              </luggage-gate-travel-time>
            </gate>
          </gates>
        </parameter>
        <parameter>
          <security xsi:type="sch:Security">
            <automatic-security-screening xsi:type="xsd:int">
              1
            </automatic-security-screening>
            <manual-security-screening xsi:type="xsd:int">
              1
            </manual-security-screening>
            <security-travel-time xsi:type="xsd:int">
              1
            </security-travel-time>
            <security-capacity xsi:type="xsd:int">
              25
            </security-capacity>
            <passport-check xsi:type="xsd:int">
              1
            </passport-check>
            <passport-travel-time xsi:type="xsd:int">
              1
            </passport-travel-time>
          </security>
        </parameter>
        <parameter>
          <luggage xsi:type="sch:Luggage">
            <automatic-luggage-screening xsi:type="xsd:int">
              1
            </automatic-luggage-screening>
            <level-2-luggage-screening xsi:type="xsd:int">
              1
            </level-2-luggage-screening>
            <luggage-travel-time xsi:type="xsd:int">
              1
            </luggage-travel-time>
            <sorting-time xsi:type="xsd:int">
              1
            </sorting-time>
          </luggage>
        </parameter>
        <parameter>
          <shopping xsi:type="sch:Shopping">
            <tourist-distribution xsi:type="xsd:int">
              0.2
            </tourist-distribution>
            <low-cost-distribution xsi:type="xsd:int">
              0.5
            </low-cost-distribution>
```

```
            <scheduled-distribution xsi:type="xsd:int">
              0.3
            </scheduled-distribution>
            <shopping-travel-time xsi:type="xsd:int">
              1
            </shopping-travel-time>
          </shopping>
        </parameter>
      </parameters>
      <outputs>
        <!--0 to 3 repetitions:-->
        <output>
          <!--You have a CHOICE of the next 3 items at this level-->
          <arrivals>
            <!--Zero or more repetitions:-->
            <arrival xsi:type="sch1:Arrival">
              <id xsi:type="xsd:int">
                ?
              </id>
              <date xsi:type="xsd:date">
                ?
              </date>
              <arrival-terminal-hour xsi:type="sch1:HourInt">
                ?
              </arrival-terminal-hour>
              <arrival-terminal-minute xsi:type="sch1:MinuteInt">
                ?
              </arrival-terminal-minute>
              <departure-flight-hour xsi:type="sch1:HourInt">
                ?
              </departure-flight-hour>
              <departure-flight-minute xsi:type="sch1:MinuteInt">
                ?
              </departure-flight-minute>
              <stay-duration-estimated xsi:type="xsd:int">
                ?
              </stay-duration-estimated>
              <check-in xsi:type="xsd:int">
                ?
              </check-in>
              <luggage-present xsi:type="xsd:int">
                ?
              </luggage-present>
              <gate-cluster xsi:type="xsd:string">
                ?
              </gate-cluster>
              <aircraft-type xsi:type="xsd:string">
                ?
              </aircraft-type>
              <flight-number xsi:type="xsd:string">
                ?
              </flight-number>
              <flight-type xsi:type="xsd:int">
                ?
              </flight-type>
              <flight-goal xsi:type="xsd:int">
                ?
              </flight-goal>
```

```
            <flight-code xsi:type="xsd:string">
              ?
            </flight-code>
            <delay-possible xsi:type="xsd:int">
              ?
            </delay-possible>
          </arrival>
        </arrivals>
        <departures>
          <!--Zero or more repetitions:-->
          <departure xsi:type="sch1:Departure">
            <id xsi:type="xsd:int">
              ?
            </id>
            <date xsi:type="xsd:date">
              ?
            </date>
            <arrival-gate-hour xsi:type="sch1:HourInt">
              ?
            </arrival-gate-hour>
            <arrival-gate-minute xsi:type="sch1:MinuteInt">
              ?
            </arrival-gate-minute>
            <departure-flight-hour xsi:type="sch1:HourInt">
              ?
            </departure-flight-hour>
            <departure-flight-minute xsi:type="sch1:MinuteInt">
              ?
            </departure-flight-minute>
            <stay-duration-actual xsi:type="xsd:int">
              ?
            </stay-duration-actual>
            <check-in xsi:type="xsd:int">
              ?
            </check-in>
            <luggage-present xsi:type="xsd:int">
              ?
            </luggage-present>
            <gate-cluster xsi:type="xsd:string">
              ?
            </gate-cluster>
            <aircraft-type xsi:type="xsd:string">
              ?
            </aircraft-type>
            <flight-number xsi:type="xsd:string">
              ?
            </flight-number>
            <flight-type xsi:type="xsd:int">
              ?
            </flight-type>
            <flight-goal xsi:type="xsd:int">
              ?
            </flight-goal>
            <flight-code xsi:type="xsd:string">
              ?
            </flight-code>
            <delay-possible xsi:type="xsd:int">
              ?
            </delay-possible>
          </departure>
```

```
            </departures>
            <flights>
              <!--Zero or more repetitions:-->
              <flight xsi:type="sch1:Flight">
                <flight-number xsi:type="xsd:string">
                  ?
                </flight-number>
                <flight-code xsi:type="xsd:string">
                  ?
                </flight-code>
                <aircraft xsi:type="xsd:string">
                  ?
                </aircraft>
                <gate-cluster xsi:type="xsd:string">
                  ?
                </gate-cluster>
                <delay-possible xsi:type="xsd:int">
                  ?
                </delay-possible>
                <date xsi:type="xsd:date">
                  ?
                </date>
                <actual-landing xsi:type="xsd:string">
                  ?
                </actual-landing>
                <actual-take-off xsi:type="xsd:string">
                  ?
                </actual-take-off>
                <scheduled-on-block xsi:type="xsd:string">
                  ?
                </scheduled-on-block>
                <actual-on-block xsi:type="xsd:string">
                  ?
                </actual-on-block>
                <boarding-start xsi:type="xsd:string">
                  ?
                </boarding-start>
                <boarding-end xsi:type="xsd:string">
                  ?
                </boarding-end>
                <forward-unload-start xsi:type="xsd:string">
                  ?
                </forward-unload-start>
                <forward-unload-end xsi:type="xsd:string">
                  ?
                </forward-unload-end>
                <forward-load-start xsi:type="xsd:string">
                  ?
                </forward-load-start>
                <forward-load-end xsi:type="xsd:string">
                  ?
                </forward-load-end>
                <aft-unload-start xsi:type="xsd:string">
                  ?
                </aft-unload-start>
                <aft-unload-end xsi:type="xsd:string">
                  ?
                </aft-unload-end>
```

```
                <aft-load-start xsi:type="xsd:string">
                  ?
                </aft-load-start>
                <aft-load-end xsi:type="xsd:string">
                  ?
                </aft-load-end>
                <scheduled-off-block xsi:type="xsd:string">
                  ?
                </scheduled-off-block>
                <actual-off-block xsi:type="xsd:string">
                  ?
                </actual-off-block>
                <pax-total xsi:type="xsd:int">
                  ?
                </pax-total>
                <pax-boarded xsi:type="xsd:int">
                  ?
                </pax-boarded>
                <pax-last-id xsi:type="xsd:int">
                  ?
                </pax-last-id>
                <pax-last-time xsi:type="xsd:string">
                  ?
                </pax-last-time>
              </flight>
            </flights>
          </output>
        </outputs>
        <!--Optional:-->
        <feedbacks>
          <!--1 or more repetitions:-->
          <feedback xsi:type="sch:Feedback">
            <step xsi:type="xsd:int">
              ?
            </step>
            <ws xsi:type="xsd:string">
              ?
            </ws>
            <tags>
              <!--1 or more repetitions:-->
              <tag>
                <name xsi:type="xsd:string">
                  ?
                </name>
                <value xsi:type="xsd:anyType">
                  ?
                </value>
              </tag>
            </tags>
            <processed xsi:type="xsd:boolean">
              ?
            </processed>
          </feedback>
        </feedbacks>
      </model>
    </experiment>
  </soapenv:Body>
</soapenv:Envelope>
```

CODE A.1: Example SOAP Request to the Airport2030 Web Service

Appendix B

XMPF UML Diagrams

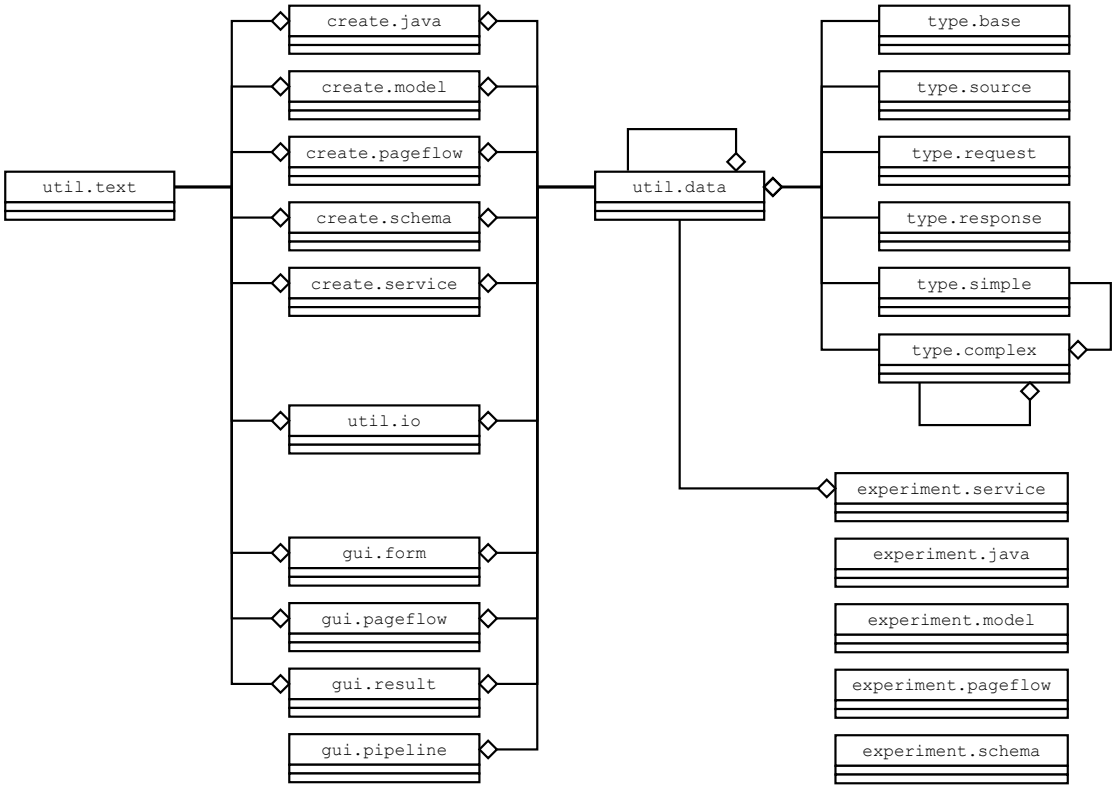


FIGURE B.1: UML Diagram of the Connections Between the XMPF Classes

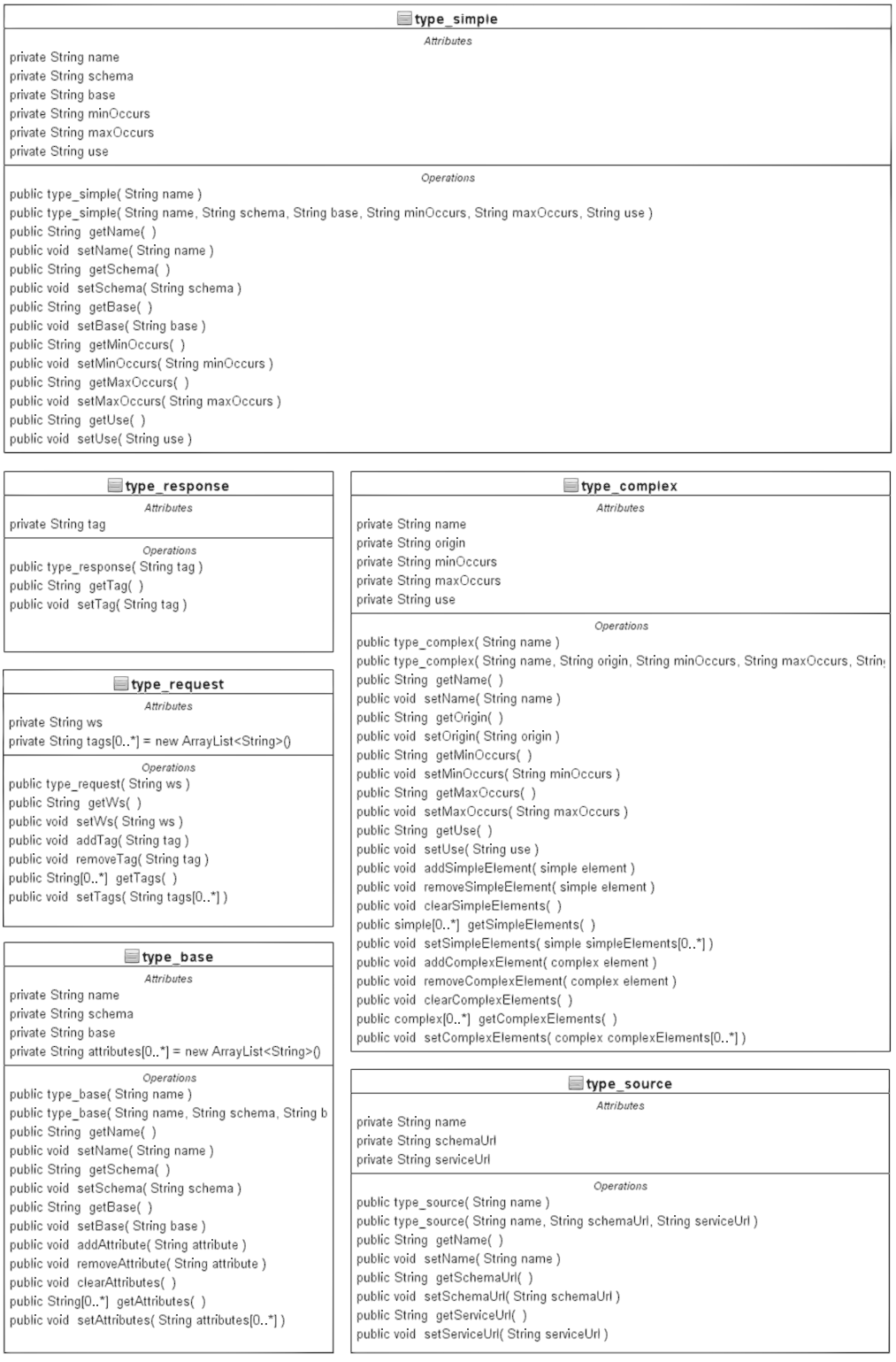


FIGURE B.2: UML Diagram of the "type" Class Package

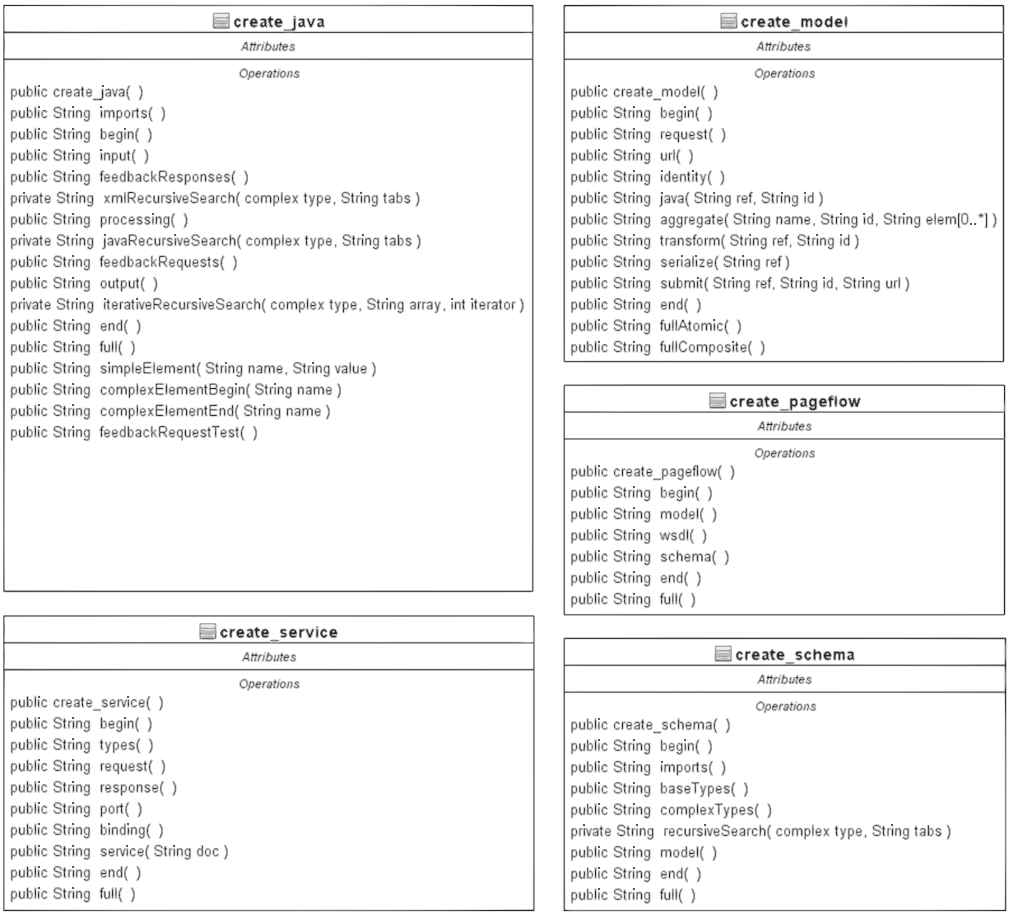


FIGURE B.3: UML Diagram of the "create" Class Package

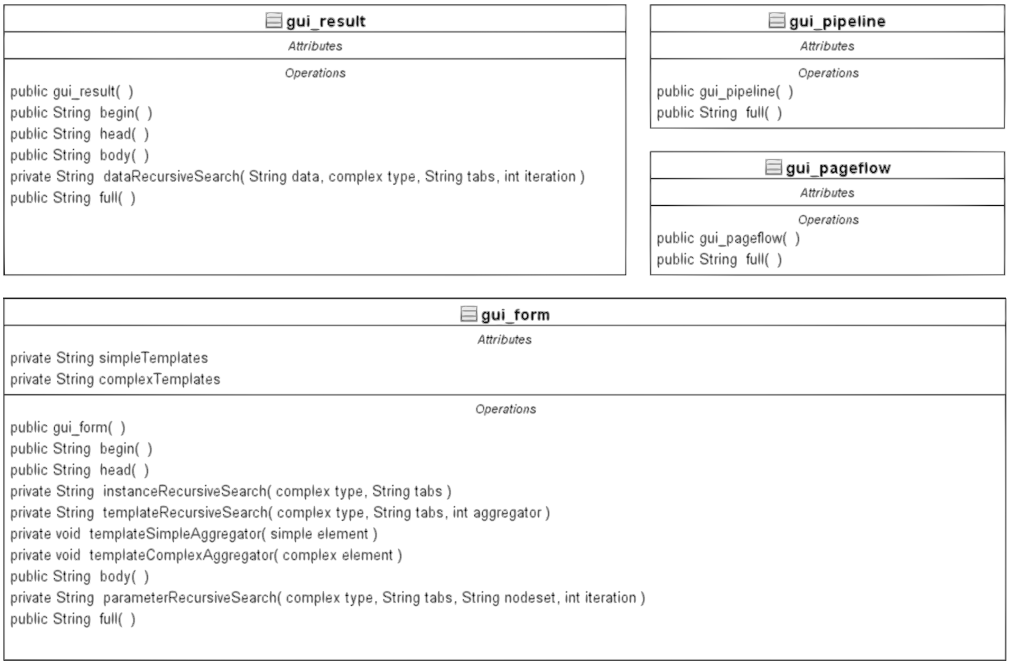


FIGURE B.4: UML Diagram of the "gui" Class Package

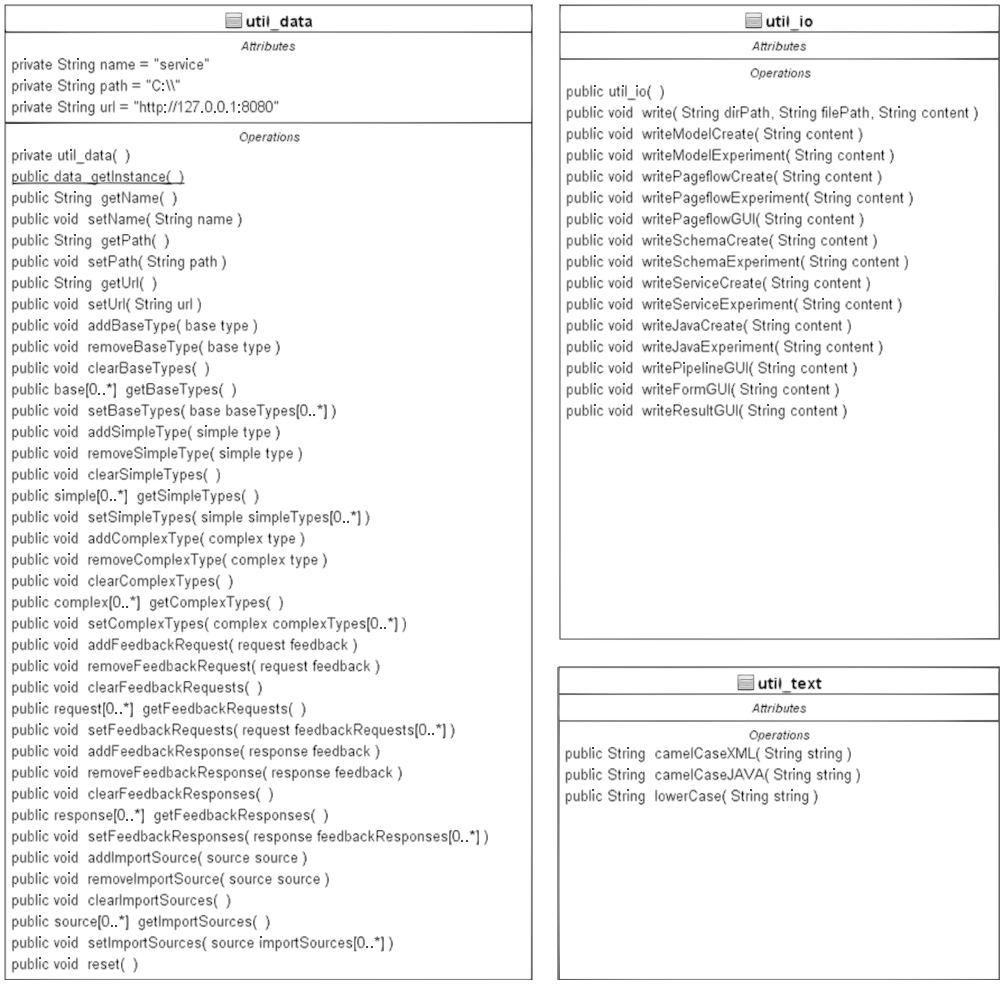


FIGURE B.5: UML Diagram of the "util" Class Package

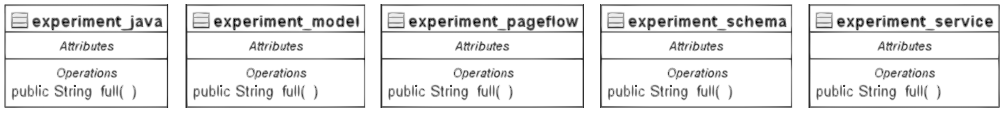


FIGURE B.6: UML Diagram of the "experiment" Class Package

Bibliography

[ACA04] ACARE. *Strategic Research Agenda 2 Vol. 1*. English. 2004. URL: http://www.acare4europe.org/docs/ASD-volume1-2nd-final-ss%20illus-171104-out-asd.pdf (visited on 06/14/2012).

[AIR09] Airport2030. *Project Plan "Airport2030"*. Mar. 2009.

[AIR11] Airport2030. *Airport2030*. May 2011. URL: http://www.airport2030.de/ (visited on 01/29/2013).

[APP99] Wolfgang Appelt. "WWW Based Collaboration with the BSCW System". In: *SOFSEM'99: Theory and Practice of Informatics*. Ed. by Jan Pavelka, Gerard Tel, and Miroslav Bartošek. Lecture Notes in Computer Science 1725. Springer Berlin Heidelberg, Jan. 1999, pp. 66–78.

[BAC00] Carliss Young Baldwin and Kim B. Clark. *Design Rules: The power of modularity*. MIT Press, 2000.

[BAC86] Maurice J. Bach. *The Design of the UNIX Operating System*. 1st ed. Prentice Hall, May 1986.

[BAN03] Jerry Banks, Jospeph C. Hugan, Peter Lendermann, Charles McLean, Ernest H. Page, C. Dennis Pegden, Onur Ulgen, and James R. Wilson. "The future of the simulation industry". In: *Proceedings of the 2003 IEEE Winter Simulation Conference*. Vol. 2. 2003, pp. 2033–2043.

[BAN98] Jerry Banks. *Handbook of simulation: principles, methodology, advances, applications and practice*. New York: Wiley, 1998.

[BBE97] A. I Bertelrud, O. Balci, C. M Esterbrook, and R. E Nance. "Developing a library of reusable model components by using the visual simulation environment". In: *Proceedings of the SSC'97*. 1997.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2003.

[BHW03] M. Bach, Kai Himstedt, and Jochen Wittmann. "Einbindung von Simulationen in computergestützte Lernumgebungen auf der Basis einer Client/Server-Architektur". In: *Simulation in Umwelt- und Geowissenschaften*. Ed. by Jochen Wittmann and Dimitris K. Maretis. Aachen: Shaker Verlag, 2003, pp. 109–128.

[BOE05] Csaba Attila Boer. *Distributed Simulation in Industry*. Rotterdam: Erasmus Research Institute of Management (ERIM), Oct. 2005.

[BRV05] Erik Bruchez and Alessandro Vernet. *XML Pipeline Language (XPL) Version 1.0 (Draft)*. 2005. URL: `http://www.w3.org/Submission/xpl/` (visited on 03/09/2011).

[BZP02] Don Brutzman, Michael Zyda, J Mark Pullen, and Katherine L Morse. *Extensible modeling and simulation framework (XMSF): Challenges for Web-based modeling and simulation*. Tech. rep. Naval Postgraduate School, Monterey, CA, 2002.

[CAR04] F. H. Carr. "C4ISR/Sim Technical Reference Model Applicability to NATO Interoperability". In: *The MITRE Corporation RTO MSG Symposium on C3I and Modeling and Simulation Interoperability*. 2004.

[CHM79] K.M. Chandy and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". In: *IEEE Transactions on Software Engineering* 5.5 (Sept. 1979), pp. 440–452.

[COM00] Douglas E. Comer. *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture*. 4th. Prentice Hall, Jan. 2000.

[COM99] S. G Cohen and D. Mankin. "Collaboration in the virtual organization". In: *Trends in organizational behavior* 6 (1999), pp. 105–120.

[COS95] L. N Cosby. *SIMNET: An Insider's Perspective.* Tech. rep. IDA DOCUMENT D-1661. Alexandria: Institute For Defense Analyses, 1995, p. 27.

[CPF99] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. "Efficient optimistic parallel simulations using reverse computation". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 9.3 (July 1999), pp. 224–253.

[DAE02] Walter F. Daenzer. *Systems Engineering: Methodik und Praxis*. Ed. by F. Huber. Zürich: Verlag Industrielle Organisation, 2002.

[DAM99] Wayne J. Davis and Gerald L. Moeller. "The high level architecture: is there a better way?" In: *Proceedings of the 31st conference on Winter simulation: Simulation - a bridge to the future*. Vol. 2. WSC '99. New York, NY, USA: ACM, 1999, pp. 1595–1601.

[DEU96] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. May 1996. URL: `http://tools.ietf.org/html/rfc1951` (visited on 08/29/2013).

[DFW97] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. "The department of defense high level architecture". In: *Proceedings of the 29th conference on Winter simulation*. 1997, pp. 142–149.

[DZG10] Niclas Dzikus and Volker Gollnick. "Modelling and Simulation of Vehicle Movements Using a SPPTW-Algorithm and the Application to Airport Surface Movement Analysis". In: *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation*. Prague, Czech Republic, Sept. 2010, p. 158.

[EVI11] eviware. *soapUI*. 2011. URL: `http://www.soapui.org/` (visited on 03/08/2011).

[EXI13] eXist Solutions. *eXist-db*. 2013. URL: `http://exist-db.org/exist/apps/homepage/index.html` (visited on 03/28/2013).

[FAN06] Rosanna Fanciulli. “Information security management and virtual collaboration: A Western Australian perspective”. In: *Proceedings of the 4th Australian Information Security Management Conference*. Edith Cowan University, Perth, Western Australia, Dec. 2006.

[FFH10] Yousef Farschtschi, Dominik Formella, Kai Himstedt, Jochen Wittmann, and Dietmar P. F. Möller. “Macroscopic Modelling of Passenger Streams on the Airport and Its Adaptation in Matlab Simulink”. In: *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation*. Prague, Czech Republic, Sept. 2010.

[FSF13] Inc. Free Software Foundation. *GNU Gzip Manual*. June 2013. URL: `http://www.gnu.org/software/gzip/manual/gzip.html` (visited on 08/29/2013).

[FUJ00] Richard M. Fujimoto. *Parallel and distributed simulation systems*. New York, NY: Wiley, 2000.

[FWH12] Yousef Farschtschi, Marc Widemann, Kai Himstedt, and Dietmar P. F. Möller. “Conceptual Design of a Two-Level Server Architecture for MATLAB-Java Coupling”. In: *Mathematical Modelling*. Vol. 7. 1. Vienna, Austria, 2012, pp. 1281–1284.

[FYW98] M. Fukunari, Yu-Liang Chi, and P.M. Wolfe. “JavaBean-based simulation with a decision making bean”. In: *Proceedings of the 1998 IEEE Winter Simulation Conference*. Vol. 2. Dec. 1998, pp. 1699–1702.

[GKN92] D. Garlan, G.E. Kaiser, and D. Notkin. “Using tool abstraction to compose systems”. In: *Computer* 25.6 (June 1992), pp. 30–38.

[GOS00] James Gosling. *Java Language Specification*. Addison-Wesley Professional, 2000.

[GSW00] A. Guru, P. Savory, and R. Williams. “A Web-based interface for storing and executing simulation models”. In: *Proceedings of the 2000 IEEE Winter Simulation Conference*. Vol. 2. 2000, pp. 1810–1814.

[HAS00] Wilhelm Hasselbring. “Information system integration”. In: *Communications of the ACM* 43.6 (June 2000), pp. 32–38.

[HHH12] Graham Hemingway, Himanshu Neema, Harmon Nine, Janos Sztipanovits, and Gabor Karsai. “Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach”. In: *SIMULATION* 88.2 (Feb. 2012), pp. 217–232.

[HIW10] Kai Himstedt and Jochen Wittmann. “Modellkopplung und Szenarioanalyse am Beispiel des Projektes "Effizienter Flughafen 2030"”. In: *Simulation in Umwelt- und Geowissenschaften*. Ed. by Jochen Wittmann and Dimitris K. Maretis. Aachen: Shaker Verlag, 2010, pp. 91–104.

[HLA00] IEEE. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules*. Sept. 2000. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=893287 (visited on 06/28/2011).

[HRX12] Heng He, Ruixuan Li, Xinhua Dong, Zhi Zhang, and Hongmu Han. “An Efficient and Secure Cloud-Based Distributed Simulation System”. In: *Applied Mathematics & Information Sciences* 6.3 (2012), pp. 729–736.

[IAT11] I.A.T.A. *Strong January Traffic Growth - Oil is the Emerging Worry*. Feb. 2011. URL: http://www.iata.org/pressroom/facts_figures/traffic_results/Pages/2011-02-28-01.aspx (visited on 04/13/2011).

[IEE00] IEEE. *IEEE 100 The Authoritative Dictionary of IEEE Standards Terms 7th Edition*. Tech. rep. Standards Information Network IEEE Press, 2000.

[ISO94] ISO. *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. Tech. rep. ISO/IEC 7498-1. ISO, Nov. 1994.

[IWG98] C4ISR Interoperability Working Group. *Levels of Information Systems Interoperability (LISI)*. Tech. rep. Washington, D.C: Department of Defense, 1998.

[JAC00] Michael Jackson. “The origins of JSP and JSD: A personal recollection”. In: *IEEE Annals of Software Engineering* 22.2 (2000), pp. 61–63.

[JEF85] David R. Jefferson. “Virtual time”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.3 (July 1985), pp. 404–425.

[KAN00] S. Kasputis and H.C. Ng. “Composable simulations”. In: *Proceedings of the 2000 IEEE Winter Simulation Conference*. Vol. 2. 2000, pp. 1577–1584.

[KAP10] J. Kaplan. *MatlabControl JMI Wrapper*. 2010. URL: `http://code.google.com/p/matlabcontrol/`.

[KIM89] Won Kim. *Object-Oriented Concepts, Databases, and Applications*. 1st ed. Association for Computing Machinery, Jan. 1989.

[KRG88] Glenn E. Krasner and Stephen T. Pope. “A cookbook for using the model-view controller user interface paradigm in Smalltalk-80”. In: *Journal of Object-Oriented Programming* 1.3 (Aug. 1988), pp. 26–49.

[KRU92] Charles W. Krueger. “Software reuse”. In: *ACM Computing Surveys (CSUR)* 24.2 (June 1992), pp. 131–183.

[KWD99] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. 1st ed. Prentice Hall, Oct. 1999.

[LAK00] Averill M. Law and W. David Kelton. *Simulation modeling and analysis*. 3rd ed. Boston, Massachussets: McGraw-Hill, 2000.

[LBB11] Sonja Löwa, Christian Blank, and Max Bohnet. “Evaluation von Einflussfaktoren auf die Verweildauer von Flugreisenden ab dem Hamburger Flughafen”. In: *Theorie und quantitative Methoden in der Geographie - Kolloquiumsbeiträge*. Vol. 19. Heidelberger geographische Bausteine. Helbich, M., Deierling, H., and Zipf, A., 2011, pp. 43–55.

[LCD05] Bo Hu Li, Xudong Chai, Yanqiang Di, Haiyan Yu, Zhihui Du, and Xiaoyuan Peng. “Research on service oriented simulation grid”. In: *Proceedings of the 5th IEEE International Symposium on Autonomous Decentralized Systems (ISADS)*. Apr. 2005, pp. 7–14.

[LER01] A. Leff and J.T. Rayfield. “Web-application development using the Model/View/Controller design pattern”. In: *Proceedings of the 5th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. 2001, pp. 118–127.

[LIP06] The Linux Information Project. *Pipes: A Brief Introduction*. Aug. 2006. URL: `http://www.linfo.org/pipes.html` (visited on 05/31/2012).

[LJD01] Simon St. Laurent, Joe Johnston, Edd Dumbill, and Dave Winer. *Programming Web Services with XML-RPC*. O’Reilly Media, Inc., June 2001.

[LOA11] Sonja Löwa. “Airport Access by Public Transport in Hamburg: Does Demand Follow Supply?” In: *WPSC World Planning School Congress 2011*. 488. Perth, WA, July 2011.

[LST00] Hau L. Lee, Kut C. So, and Christopher S. Tang. “The Value of Information Sharing in a Two-Level Supply Chain”. In: *Management Science* 46.5 (May 2000). ArticleType: research-article / Full publication date: May, 2000 / Copyright © 2000 INFORMS, pp. 626–643.

[MAT11] Mathworks. *MathWorks - MATLAB and Simulink for Technical Computing*. 2011. URL: `http://www.mathworks.com/` (visited on 06/30/2011).

[MBN68] M. D. McIlroy, J. M. Buxton, P. Naur, and B. Randell. “Mass-produced software components”. In: *Software Engineering; Report on a conference by the NATO Science Committee* NATO Scientific Affairs Division (1968). Ed. by P. Nauer and B. Randell, pp. 138–150.

[MCI64] M. D. McIlroy. *Pipes and filters*. 1964.

[MCL02] Timothy J. McLaughlin. *The Benefits and Drawbacks of HTTP Compression*. Tech. rep. 19. Department of Computer Science and Engineering, Lehigh University, 2002.

[MIT95] D. C Miller and J. A Thorpe. “SIMNET: the Advent of Simulator Networking”. In: *Proceedings of the IEEE* 83.8 (Aug. 1995), pp. 1114–1123.

[MOD06] B. Möller and C. Dahlin. “A first look at the HLA evolved Web service API”. In: *Proceedings of the 2006 Euro Simulation Interoperability Workshop*. 2006.

[MOL06] B. Möller and S. Löf. “A management overview of the HLA evolved web service API”. In: *Proceedings of the 2006 Fall SISO Simulation Interoperability Workshop*. 2006.

[MOP06] Dietmar P. F. Möller and Hortensia Popescu. “HLA Simulator For Land Based Transportation”. In: *SCS Conference 2000*. Ed. by D. W. Waite. San Diego, 2006, pp. 704–709.

[MWK12] Björn Möller, Åsa Wihlborg, Filip Klasson, Mikael Karlsson, Steve Eriksson, Patrik Svensson, Björn Löfstrand, Martin Johansson, and Pär Aktanius. *The HLA Tutorial - A Practical Guide for Developing Distributed Simulations*. 2012. URL: `http://sisosmackdown.com/wp-content/uploads/2011/07/The-HLA-Tutorial.pdf` (visited on 05/18/2013).

[NHT04] S. K. Numrich, M. Hieb, and A. Tolk. *M&S in the GIG environment: An expanded view of distributing simulation*. Tech. rep. DTIC Document, 2004.

[NLJ96] H. S. Nwana, L. C. Lee, and N. R. Jennings. “Coordination in software agent systems”. In: *British Telecom Technical Journal* 14.4 (1996), pp. 79–88.

[OAS02] OASIS. *OASIS UDDI Specifications TC-Committee Specifications*. 2002. URL: `https://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm` (visited on 03/09/2011).

[OEC80] T. I Ören and B. Collie. “Design of SEMA: A software system for computer-aided modelling and simulation of sequential machines”. In: *Proceedings of the 12th conference on Winter simulation*. 1980, pp. 113–123.

[OMG12] Object Management Group. *OMG Formal Specifications*. 2012. URL: `http://www.omg.org/spec/` (visited on 10/11/2012).

[ORB11] Orbeon. *Orbeon Forms - Web Forms for the Enterprise, Done the Right Way*. Mar. 2011. URL: `http://www.orbeon.com/` (visited on 05/17/2011).

[PAH03] Mike P Papazoglou and Willem-Jan van den Heuvel. “Service-Oriented Computing: State-of-the-Art and Open Research Issues”. In: *IEEE Computer. v40 i11* (2003).

[PAR72] D. L. Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058.

[PBT04] E. H. Page, R. Briggs, and J. A. Tufarolo. “Toward a family of maturity models for the simulation interconnection problem”. In: *Proceedings of the Spring Simulation Interoperability Workshop*. 2004.

[PCH93] J. S. Poulin, J. M. Caruso, and D. R. Hancock. “The business case for software reuse”. In: *IBM Systems Journal* 32.4 (1993), pp. 567–594.

[PEL03] C. Peltz. “Web services orchestration and choreography”. In: *Computer* 36.10 (2003), pp. 46–52.

[PER05] Anders Persson. “Migration and Load balancing in HLA based distributed simulations”. PhD thesis. Sweden: Stockholm, 2005.

[PER06] K.S. Perumalla. “Parallel and Distributed Simulation: Traditional Techniques and Recent Advances”. In: *Proceedings of the 2006 Winter Simulation Conference (WSC)*. 2006, pp. 84–95.

[PEW03] M. D. Petty and E. W. Weisel. “A composability lexicon”. In: *Proceedings of the 2003 Spring Simulation Interoperability Workshop*. 2003, pp. 181–187.

[PHL09] Peter Phleps. *Szenarioerstellung im Rahmen des Leuchtturmprojektes "Effizienter Flughafen"*. Deutsch. 2009.

[PIT10] Pitch. *Pitch Visual OMT version 2*. 2010. URL: `http://www.pitch.se/images/files/productsheets/VisualOMTv2.pdf`.

[POB99] M. Pidd, N. Oses, and R.J. Brooks. “Component-based simulation on the Web?” In: *Proceedings of the 1999 Winter Simulation Conference*. Vol. 2. 1999, pp. 1438–1444.

[PRL89] Bruno R. Preiss and Wayne M. Loucks. *Prediction and Lookahead in Distributed Simulation*. Report. Computer Communications Group, University of Waterloo, 1989.

[RAR03] T. Ravichandran and Marcus A. Rothenberger. "Software reuse strategies and component markets". In: *Communications of the ACM* 46.8 (Aug. 2003), pp. 109–114.

[RAT93] Hassan Rajaei, Rassul Ayani, and Lars-Erik Thorelli. "The local Time Warp approach to parallel simulation". In: *Proceedings of the 7th workshop on Parallel and distributed simulation* 23.1 (July 1993), pp. 119–126.

[REE79] Trygve Reenskaug. *Thing-Model-View-Editor an Example from a planningsystem*. Technical Note. Xerox PARC, May 1979.

[RIC95] L. Ricciulli. "A technique for the distributed simulation of parallel computers". In: *Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Jan. 1995, pp. 257–260.

[ROL10] V. Roso and K. Lumsden. "A Review of Dry Ports". In: *Maritime Economics & Logistics*. Vol. 12. 2010, pp. 196–213.

[RTR98] Dhananjai Madhava Rao, Narayanan V. Thondugulam, Radharamanan Radhakrishnan, and Philip A. Wilsey. "Unsynchronized parallel discrete event simulation". In: *Proceedings of the 30th conference on Winter simulation*. WSC '98. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 1563–1570.

[SCN96] R. W. Schulte and Y. V. Natis. *Service oriented'architectures, part 1*. Research Note SPA-401-068. Gartner, 1996.

[SGM95] W3C. *Standard Generalized Markup Language (SGML)*. 1995. URL: `http://www.w3.org/MarkUp/SGML/` (visited on 03/09/2011).

[SHG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.

[SLA04] Erick D. Slazinski. “Structured Query Language (SQL)”. In: *The Internet Encyclopedia*. John Wiley & Sons, Inc., 2004.

[SLM01] D. Sly and S. Moorthy. “Simulation data exchange (SDX) implementation and use”. In: *Proceedings of the 2001 Winter Simulation Conference*. Vol. 2. 2001, pp. 1473–1477.

[SOA07] W3C. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. 2007. URL: `http://www.w3.org/TR/soap12-part1/` (visited on 03/09/2011).

[SOM12] Ian Sommerville. *Software Engineering*. 9., aktualisierte Aufl. München [u.a.]: Pearson, 2012.

[SSK99] Thomas Schulze, Steffen Strassburger, and Ulrich Klein. “Migration of HLA into Civil Domains: Solutions and Prototypes for Transportation Applications”. In: *SIMULATION* 73.5 (Nov. 1999), pp. 296–303.

[SSL07] Steffen Strassburger, T. Schulze, and M. Lemessi. “Applying CSPI reference models for factory planning”. In: *Proceedings of the 39th Conference on Winter Simulation: 40 years! The best is yet to come*. 2007, pp. 603–609.

[STR06] Steffen Strassburger. “Overview about the High Level Architecture for Modelling and Simulation and Recent Developments”. In: *Simulation News Europe*. Parallel and Distributed Simulation Methods and Environments 16.2 (Sept. 2006), pp. 5–14.

[SVG11] W3C. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. Aug. 2011. URL: `http://www.w3.org/TR/SVG/` (visited on 02/07/2013).

[TBB09] M. Turner, D. Budgen, and P. Brereton. “Turning software into a service”. In: *Computer* 36.10 (Oct. 2009), pp. 38–44.

[TFC06] W. T. Tsai, Chun Fan, Yinong Chen, and Ray Paul. “DDSOS: A Dynamic Distributed Service-Oriented Simulation Framework1”. In: *Proceedings of the 39th annual Symposium on Simulation*. ANSS ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 160–167.

[TMS07] S. J. E. Taylor, N. Mustafee, S. Strassburger, S. J. Turner, M. Y. H. Low, and J. Ladbrook. "The SISO CSPI PDG standard for commercial off-the-shelf simulation package interoperability reference models". In: *Proceedings of the 39th Conference on Winter Simulation: 40 years! The best is yet to come*. 2007, pp. 594–602.

[TOC63] Keith Douglas Tocher. *The Art of Simulation*. London: English Universities Press, 1963.

[TOH90] Frederich N. Tou and Wagar Hasan. "Method for integrating a knowledge-based system with an arbitrary database ..." Pat. 4930071. U.S. Classification: 1/1 International Classification: : G06F 1520. May 1990. URL: `http://www.google.com/patents?id=FxcbAAAAEBAJ` (visited on 10/08/2012).

[TOM03] A. Tolk and J. A. Muguira. "The levels of conceptual interoperability model". In: *Proceedings of the 2003 Fall Simulation Interoperability Workshop*. Vol. 7. 2003.

[TUR05] C.D. Turnista. "Extending the Levels of Conceptual Inter-operability Model". In: *Proceedings of the 2005 IEEE Summer Computer Simulation Conference*. IEEE CS Press, 2005.

[TUT99] J. R. Turner and B. Taylor. *The handbook of project-based management*. Vol. 92. McGraw-Hill London, 1999.

[UML12] Object Management Group. *Unified Modeling Language (UML)*. 2012. URL: `http://www.omg.org/spec/UML/` (visited on 10/21/2013).

[WAD04] L. Wainfan and P. K. Davis. *Challenges in virtual collaboration: Videoconferencing, audioconferencing, and computer-mediated communications*. Vol. 273. RAND Corporation, 2004.

[WEE05] Sanjiva Weerawarana. *Web services platform architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more*. Upper Saddle River, NJ: Prentice Hall, 2005.

[WFH11] Marc Widemann, Yousef Farschtschi, Kai Himstedt, and D. P. F. Möller. "Prototypische Umsetzung einer Modellpipeline mit Rückkopplung". In: *Simulation in Umwelt- und Geowissenschaften Workshop Berlin 2011*. Ed. by Jochen Wittmann and Volker Wohlgemuth. Aachen: Shaker Verlag, 2011, pp. 165–178.

[WFH12] M. Widemann, Y. Farschtschi, K. Himstedt, D. P. F. Möller, and J. Wittmann. "A Concept to Enhance the Feedback Capabilities of Model Pipelines". In: *Mathematical Modelling*. Ed. by F. Breitenecker and I. Troch. Vol. 7. 1. Austria, Vienna, Feb. 2012, pp. 1289–1293.

[WHM09] Jochen Wittmann, Kai Himstedt, and D. P. F. Möller. "Ein Pipeliningkonzept zur Modellierung der Passagier und Gepäckbearbeitung am Flughafen". In: *Simulationstechnik/Simulation Techniques 20. Symposium in Cottbus*. Ed. by Albrecht Gnauck and Bernhard Luther. Aachen: Shaker Verlag, 2009, pp. 404–414.

[WSA09] W3C. *Web Services Activity*. 2009. URL: `http://www.w3.org/2002/ws/#documents` (visited on 03/09/2011).

[WSD01] W3C. *Web Services Description Language (WSDL) 1.1*. 2001. URL: `http://www.w3.org/TR/wsdl` (visited on 03/09/2011).

[WWH10] Marc Widemann, Jochen Wittmann, Kai Himstedt, and Dietmar P. F. Möller. "Macroscopic Modeling of a Luggage Stream in an Airport Terminal". In: *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation*. Prague, Czech Republic, Sept. 2010.

[WYL08] W. Wang, W. Yu, Q. Li, W. Wang, and X. Liu. "Service-oriented high level architecture". In: *Proceedings of the 2008 Summer Computer Simulation Conference*. 2008, p. 16.

[XFO03] W3C. *XForms 1.0 Frequently Asked Questions*. 2003. URL: `http://www.w3.org/MarkUp/Forms/2003/xforms-faq.html` (visited on 02/06/2013).

[XFO09] W3C. *XForms 1.1*. 2009. URL: `http://www.w3.org/TR/xforms/` (visited on 03/09/2011).

[XHT10] W3C. *XHTML 1.1 - Module-based XHTML - Second Edition*. Nov. 2010. URL: http://www.w3.org/TR/xhtml11/ (visited on 03/28/2013).

[XML00] W3C. *XML Schema*. 2000. URL: http://www.w3.org/XML/Schema.html (visited on 04/06/2011).

[XML08] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Nov. 2008. URL: http://www.w3.org/TR/REC-xml/ (visited on 05/17/2011).

[XPA99] W3C. *XML Path Language (XPath)*. Nov. 1999. URL: http://www.w3.org/TR/xpath/ (visited on 06/14/2011).

[XSL99] W3C. *XSL Transformations (XSLT)*. 1999. URL: http://www.w3.org/TR/xslt (visited on 04/06/2011).

[YAG97] Susan E. Yager. "Everything's coming up virtual". In: *Crossroads* 4.1 (Oct. 1997), pp. 20–24.

[ZAD94] Lotfi A. Zadeh. "Fuzzy logic, neural networks, and soft computing". In: *Communications of the ACM* 37.3 (Mar. 1994), pp. 77–84.

[ZEI87] Bernard P Zeigler. "Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment". In: *SIMULATION* 49.5 (Jan. 1987), pp. 219–230.

[ZSK95] Bernard Zeigler, Hae Song, Tag Kim, and Herbert Praehofer. "DEVS framework for modelling, simulation, analysis, and design of hybrid systems". In: *Hybrid Systems II*. Ed. by Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry. Vol. 999. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1995, pp. 529–551.